The Pragmatic Programmers

TURING 图灵程序设计丛书

Lean from the Trenches

Managing Large-Scale Projects with Kanban

精益开发实战

用看板管理大型项目

【瑞典】Henrik Kniberg 著

李祥青 译



- 剖析实际项目，揭示敏捷和精益方法精髓
- 手把手教你用看板/Scrum玩转大型项目
- 《硝烟中的Scrum和XP》作者新作



人民邮电出版社

POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

Lean from the Trenches
Managing Large-Scale Projects with Kanban

精益开发实战
用看板管理大型项目

【瑞典】Henrik Kniberg 著
李祥青 译

人 民 邮 电 出 版 社
北 京

图书在版编目（ＣＩＰ）数据

精益开发实战 : 用看板管理大型项目 / (瑞典) 克里伯格 (Kniberg,H.) 著 ; 李祥青译. -- 北京 : 人民邮电出版社, 2012.9
(图灵程序设计丛书)
书名原文: Lean from the Trenches: Managing Large-Scale Projects with Kanban
ISBN 978-7-115-29177-6

Ⅰ. ①精… Ⅱ. ①克… ②李… Ⅲ. ①项目管理 Ⅳ. ①F27

中国版本图书馆CIP数据核字(2012)第188245号

内 容 提 要

本书从实践角度展示如何使用看板管理大型项目。书中内容共分为两大部分。第一部分是案例研究，讲述看板和精益原则在具体项目中的运用；第二部分是技术详解，详细介绍第一部分提到的因果图等实践做法。

本书适合软件开发组织中的项目团队主管、经理和其他变更负责人，也适合一切对敏捷开发感兴趣的人士。

图灵程序设计丛书

精益开发实战：用看板管理大型项目

- ◆ 著 [瑞典] Henrik Kniberg
  译 李祥青
  责任编辑 朱 巍
- ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
  邮编 100061 电子邮件 315@ptpress.com.cn
  网址 http://www.ptpress.com.cn
  北京 印刷
- ◆ 开本：700×1000 1/16
  印张：11.5
  字数：153千字 2012年9月第1版
  印数：1－4 000册 2012年9月北京第1次印刷

著作权合同登记号 图字：01-2012-5030号

ISBN 978-7-115-29177-6

定价：39.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223

反盗版热线：(010)67171154

版权声明

对本书的赞誉

“太棒了！我就知道这本书会在软件开发领域产生重大影响。这是过去一年中最重要的一本书!”

——玛丽·帕彭迪克（Mary Poppendieck），
《精益软件开发》系列书籍作者

“《精益开发实战》让我无法释手。这本书介绍了如何以精益而敏捷的方式运行大型项目。对于正处于实战中的大型企业人员来说，这种成功案例意义深远。”

——伊夫斯·哈诺（Yves Hanoulle），PairCoaching.net 变更大师

“很震撼！终于出现了一个思路真实可用、不干巴的、注重实效的成功案例。”

——西蒙·克罗默蒂（Simon Cromarty），
敏捷海盗网（http://theagilepirate.net）

“这本书讲述用敏捷和精益原则管理真实项目，我非常喜欢，书的内容非常实用，可读性极高。强调真实经验而非理论让人耳目一新，非常吸引人。我绝对要向朋友们推荐这本书，而且会将书中的独到见解运用到我自己的专业咨询项目中去。”

——凯文·毕姆（Kevin Beam），Lambda42 公司独立软件开发工程师

序

作为项目咨询师，我们面对着强大的诱惑。请我们提供咨询服务的团队期待的是方向、希望、构想和指导（有时是想找个替罪羊，不过那暂且不论）。之所以有人求助于我们，是因为我们见多识广，好的坏的情况都遇到过。我们的工作则尽全力帮助客户朝着好的方向前进。不过，接下来究竟该做什么，我们往往跟客户一样困惑。

我说的诱惑是指不顾经验教训想信口开河，为满足客户需要的确定性而自欺欺人地编造确定性。如果放任自己不加约束，最终就会陷入教条主义，满嘴“必须”、“始终”和“任何人”等绝对性的词汇。

本书所述案例的一大优点就是摒弃一切教条。书中讲了一个故事，这个项目遇到了真实的问题，而项目团队运用一些易于理解的实践做法去解决了这些问题。这些实践做起来需要一定的智慧和耐心，还要坚持不懈，也正因如此，您不能照搬此案例去拯救自己手中的项目。

不能照搬此案例的另一个原因是，本书的目的并不是提供一份通用的处方。这个案例有着特定的客户和处于特定文化之下的特定团队。您需要结合自身的条件灵活运用书中的知识。不过这样也好，因为无论如何你都肯定会遇到各式各样的环境。

您会看到如何运用通用的原则。我曾有幸与亨瑞克一起共事，他亲口告诉我他真的只有一个秘诀：将所有重要信息集中在一起展示，然后大家一起决定该做什么。如果这真是他唯一的秘诀（我可不太相信），那也是非常管用的一个秘诀。

社会已经学会不再信任我们复杂且最终毫无用处的大型公共软件项目。而本书则讲述了一个最终成功服务于公众的公共服务项目。要赢得公众的信任，我们需要的是团队协作、透明性、尽早以及频繁的版本发布。哎哟，我自己就没能经得住前面才跟您说的诱惑。您还是亲自读一读这本书，自己去体会经验教训吧。

肯特·贝克
2011 年 9 月

前 言

很多人都听说过精益软件开发、看板和其他一些时髦的术语。不过，这些东西实际上究竟是什么样？如果一个项目是开发真实的复杂系统，项目团队扩展到60人时应该如何管理？

我无法告诉你具体该怎么做，因为每个项目的具体情况都会不同。不过我可以告诉你我们是如何做的（基本上是结合了Scrum、XP和看板），或许其中一些解决方案以及我们得到的经验教训能够为你所用。

读者对象

本书的主要读者是软件开发组织中的项目团队主管、经理、教练和其他变更负责人。

不过，本书的部分内容对于有兴趣了解软件开发、精益产品开发或者常用的协作技术知识的读者，可能都有用，无论你是什么角色或处于什么行业。

如果你希望对本书加以评论，请前往本书主页，访问论坛和勘误页面。欢迎你提出宝贵意见！

如何阅读本书

本书由两部分组成，每个部分都分成多个篇幅短小的章。

第一部分“我们如何工作”是案例研究，讲述我们如何将看板和精益原则运用到为瑞典国家警署开发的大型项目中。第 1 章介绍了项目概况，

接下来的各章详细介绍了我们面临的具体挑战（如项目需要扩展）、我们如何处理这些挑战，以及我们得到的经验教训。

第二部分“技术详解”首先概要介绍了敏捷和精益，然后详细介绍本书第一部分中提到的一些实践做法，比如因果图。

建议你先全文阅读第一部分，因为这是本书的核心所在，而且其中所有章的内容都互相关联。然后，再从第二部分中挑选自己感兴趣的内容，因为这部分各章的内容相对独立。

第一次接触敏捷或精益？

如果你是第一次接触敏捷或精益思想，也别担心，本书的内容面向实践，而不是理论。我只会告诉你我们的具体做法，你读的时候就会对相关理论有概念了。

如果你想先初步了解一下敏捷和精益思想以及相关的方法（Scrum、XP和看板），那么可以直接阅读第 17 章。

免责声明

我并未说我们的工作方式是完美的精益方式。精益是方向，而不是目的。精益的核心是持续改进。精益没有明确的定义，但我们应用的很多精益实践都是以玛丽・帕彭迪克、大卫・安德森和唐・类纳森所教授的精益产品开发原则为基础的。而且，在多数情况下，这些精益实践都可与敏捷原则完美匹配。

另外，你只是从我的视角看这个项目。我是项目的兼职教练，参与了六个月的时间。我没有要去展示项目 100%的完整面貌，而是想让你了解我们具体做了什么，以及目前为止得到了什么经验教训。

致谢

很多人都对这本书的出版作出了贡献，感谢所有的人！我想特别感谢哈贝·雷德曼，他作为内部变更负责人将我带入项目，感谢托马斯·阿尔斯特尔伦德强有力的管理，让我们可以专注于项目目标。

我还想向以下这些人致谢。

克里斯蒂安·斯图尔特以及 RPS 管理团队的其他人，感谢他们委托我担任这一项目的教练，并允许我们宣传自己的具体做法。

项目所有参与人员，感谢他们全心投入到此项目中，帮助推动变更流程。我为项目团队所拥有的高超技能、独特创造力和巨大能量深深折服！

玛丽与汤姆·帕彭迪克夫妇，感谢他们多年来对我在精益软件开发方面给予的教诲与培训，并鼓励我写了这本书。他们还慷慨地提供了 17.2 节中的大部分内容。

我的编辑凯·克普勒。在此之前我还从未与编辑合作过，此次合作经历弥足珍贵，让我受益匪浅。凯不但帮助润色了本书的文字，还帮助我提高了写作水平！

本书所有审校人员：Gunnar Ahlberg、Kevin Beam、Kent Beck、Pawel Brodzinski、Ward Cunningham、Doug Daniels、Chad Dumler-Montplaisir、Yves Hanoulle、Michael Hunter、Andy Keffalas、Maurice Kelly、Sebastian Lang、Rasmus Larsson、Mary Poppendieck、Sam Rose、Daniel Teng、Nancy Van Schooenderwoert、Joshua White 与 Colin Yates。

马提·史密斯和艾玛·马特森，感谢他们免费提供部分珍贵的照片。

最后感谢我的妻子索非亚，是她让我专注工作（带着四个小孩子，真是不易），我才能用了数天，而不是数月就完成了本书的初稿。

亨瑞克·尼伯（Henrik Kniberg）
电子邮件：henrik.kniberg@crisp.se
2011 年 10 月于斯德哥尔摩

目　　录

第一部分　我们如何工作

第二部分　技术详解

Part 1

第一部分

我们如何工作

让我们一起进入实战状态，看看这是一个什么样的项目以及此项目具体是如何管理的。

第1章

项目背景

瑞典国家警署是瑞典国家警察机关。我们开发的产品是一个全新的名为 PUST（Polisens mobila Utrednings Stöd）的数字调查系统。项目的基本理念是为每台警车配备一台小型手提电脑，具备移动连网功能，以及一个网页应用，来让警察快速处理调查工作。

假设警察抓到一名醉驾司机。过去，警察需要做笔录记下所有信息，开车到警局，提交报告，然后把案子移交给另一位调查员继续进行调查。处理这样一起案子需要一个月左右的时间。

使用 PUST 后，警察就可以将所有信息直接输入手提电脑查询资料，因为手提电脑有无线网络连接，且与所有相关系统直接集成。这个案子只需要几天甚至几小时就可以结案。



我们的调查系统于 2011 年 4 月开始在瑞典全国范围推广，受到媒体高度关注。各大报纸、电视、广播都对我们的产品做了专题报道，而且到目前为止获得了非常积极的反响 。



NYHETER

Nytt utredninssystem underlättar polisens arbete

DN.

Antal dagar från brottsplats till åklagare

Polisen utreder betydligt snabbare med ny metod

Dagens industri

LEAN

LÄGG GRUNDEN FÖR LEAN

Länspolismästare Christina Forsberg: Min grundinställning är att alla som får rätt förutsättningar kan prestera mer och att man kan skapa effektivitet genom delaktighet.

Polisen jobbar agilt Så får de stöd i arbetet av nytt utredningssystem

OEE/TAK "Systemet möjliggör ett snabbare förbättringsarbete"

现在警察处理轻度犯罪案件的效率平均比以前快六倍 ，这样他们有更多的时间出警，而不是呆在警局处理案头工作。警察可以更到位地处理更多犯罪案件，从长远看有助于降低犯罪率。警察出警很积极，因为他们喜欢做警察该做的工作，而不是整天做案头工作！

此外，项目本身进展非常顺利。让人惊异的是，与以往一些复杂度和规模类似的旧项目相比，我们收到的售后支持问题和 Bug 报告非常之少。PUST 系统之所以复杂是因为它有以下特点：

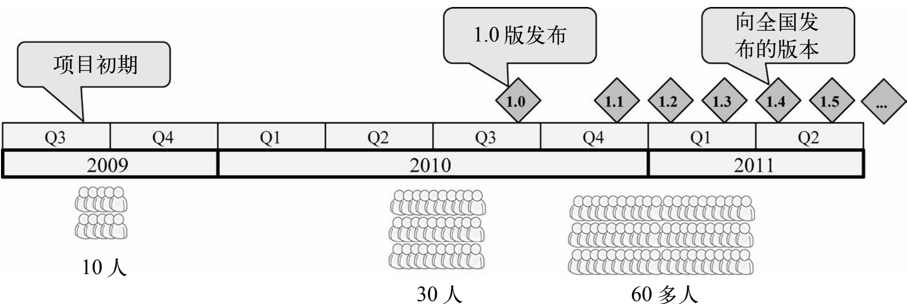
- 与大量的遗留系统集成；
- 用户体验必须非常好，因警察需要一边询问犯罪嫌疑人一边使用系统；
- 安全性要很高；
- 必须符合大量法律法规的规定。

此项目对瑞典警察总署非常重要。实际上，司法部部长已公开宣称瑞典警察的工作重点是提高效率，缩短案件调查处理时间。风险高、技术复杂、时间紧，都让我们很清楚，运用传统方法绝无成功可能。于是，我们获准开始探索新的、更高效的工作方式，这就是本书要为你讲述的内容。

PUST 是瑞典警察总署内部文化变革的一部分，是一项涉及整个组织的全国性精益举措。所以，将精益原则应用于系统开发过程本身也就顺理成章，合乎情理！

1.1 时间线

这个项目的目标是打算到 2011 年早期，就能让瑞典所有警察都使用。开发工作大约从 2009 年 9 月开始。第一个生产版本（试点）一年后发布，然后是每两个月发布一个更新版本。



项目团队在 2009 年第三季度的最初规模是 10 人，2010 年中期扩大到 30 人，然后到 2010 年第四季度的时候规模翻倍，达到 60 多人。

关键的里程碑是 1.0 版（向真正用户发布的首个试点版本）和 1.4 版（向全国发布的版本）。当然，PUST 系统在以后的多年时间里会继续改进，发布更新版本，所以 1.4 版绝不会是最后一个版本。

或许很多敏捷开发人士会说一年才发布第一个版本似乎太久，但是与类似复杂度和规模的其他政府项目相比，这个时间已非同一般地短。曾有

类似的政府项目开发了长达七年之久才发布第一个版本！ 而且我们每两个月都发布更新版本的做法也不寻常，很多政府组织通常一年才发布一两个更新版本，我们甚至都在考虑每个月都发布。

小乔爱问：
为什么频繁发布？这样成本岂不是很高？

的确，每个发布版本都有一定的固定成本。但是每个发布版本都是了解真相的最佳时刻——我们这时才能了解产品是否与用户的需求契合！两次发布的间隔时间越长，我们在代码中嵌入的缺陷和错误假设就会越多。如果发布频率高一些，补丁小一些，每个版本的问题和风险也就相应降低了许多。

所有这些因素——发布周期短以及项目范围迅速扩大——都成为组织和开发流程需要快速改进的动因。

这也是我作为教练参与到这个项目中的缘由。

我从 2010 年 12 月到 2011 年 6 月期间参与了这个项目，每周在项目上的工作时间大概是两到三天。我的工作重点是将精益和敏捷原则投入实践，帮助开发团队制定最合身的流程。这就是本书要为你讲述的内容——我们做了什么、遇到了什么问题、如何处理这些问题，以及我们得到了哪些经验教训。这一段经历挑战性极强，却也乐趣无穷！

不过，需要牢记的一点是……

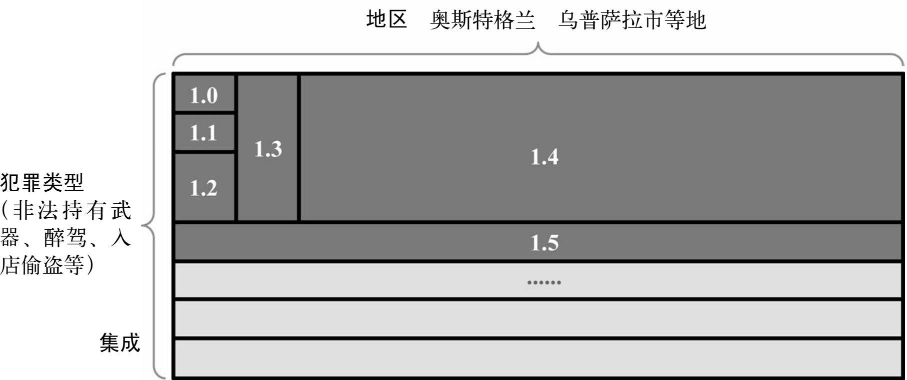
这本书基本上是项目流程 2011 年 6 月时的一个快照。精益流程一个最重要的特点就是不断发展改进。有时我们发现了更好的解决方案；有时昨天看上去相当不错的解决方案今天就会带来新问题；有时环境变了，迫使我们做出改变来适应新环境。

所以，等你读到这本书的时候，我们的项目可能已经是另一副模样了。

1.2 我们如何切割大象

最大程度降低大型项目风险的关键是找到“切割大象”的正确方式，也就是说，找到以小幅递增模式发布系统的合适方式，而不是将所有开发成果积累到一起，最后才来个让人措手不及的爆炸性发布。最理想的情况是，每次的增幅能够独立地为用户带来价值，为团队积累知识。

我们从两个维度切割这头大象：地理区域和犯罪类型。

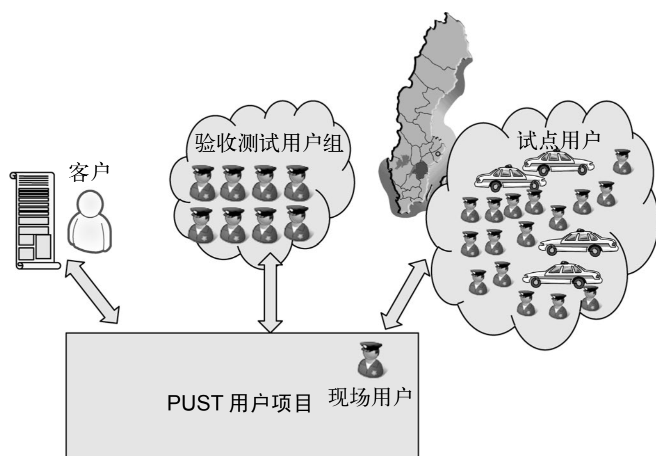


- 1.0 版~1.2 版：只向一个地区（奥斯特格兰）发布的试点版本，仅支持几个常见犯罪类型，如醉驾和非法持有武器。其他类型的犯罪还是以旧的手动方式处理。在接下来的每一个更新版本中，我们都改进系统稳定性、增加支持的犯罪类型。
- 1.3 版：增加第二个地区（乌普萨拉市）。
- 1.4 版：将系统扩展到整个瑞典。这就是“主要”版本。
- 1.5 版：增加犯罪类型，与其他系统（如查抄物品跟踪系统）集成。

除了每两个月一次的更新版本外，我们每隔几周还会发布“补丁”版本，提供 Bug 修复和对现有功能的小改进。

1.3 我们如何让客户参与进来

PUST 是内部开发项目，客户、用户和开发人员都属于瑞典警察机关。



有专人担任项目的主要“客户”（或“买家”），她为我们列出概要功能清单。我们称之为*功能区域*（feature areas），大致等同于敏捷开发人员熟悉的*综述*（epics）概念。这个功能清单用于制订概要时间表和版本发布计划。

除客户以外，还有*现场用户*与开发团队在一起。现场用户的任务是提供详细反馈、观看产品功能演示、回答开发人员的问题等。最初现场用户大概每周只来一次，不过到后期我们几乎每天都有轮班现场用户。

每次发布前一周，我们都会有一个*验收测试*用户组过来，通常是十名左右的警察、调查官或其他真实用户。这个用户组会用几天时间来试用最新的发布候选版并提供反馈。等到要进行验收测试的时候，系统状态通常都已经相当好，所以到这个时候很少会出现让人紧张的意外情况。

等首个版本一发布出去，我们就会有一组*试点用户*开始在奥斯特格兰（瑞典南部一地区）开始辛勤工作，连续不断地为我们的产品提供反馈。

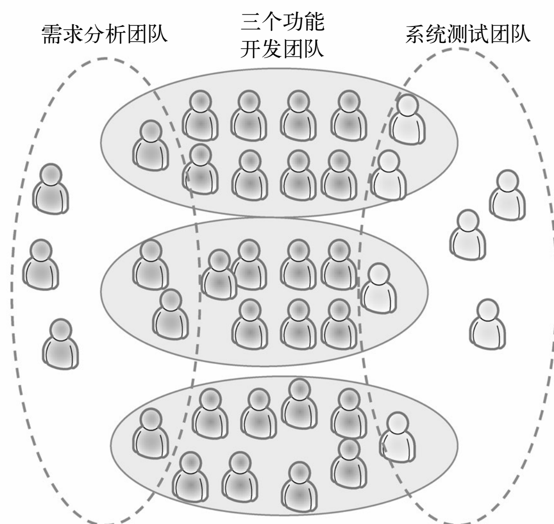
第2章

组织团队

软件开发项目的一大挑战是如何将项目人员划分成多个大小适中的团队，并让这些团队相互紧密合作。

随着项目团队从 30 人增加到 60 多人，我们开始遇到严重的沟通和协调问题，这是成长之痛的典型症状。所幸我们都坐在同一层楼内，所有项目人员彼此之间最远走过去也就是半分钟的距离。所以，我们可以毫无困难地开始实验如何以最佳的方式组织项目团队。实际上，项目团队在一起办公也许是此项目成功的最重要因素。

我们的团队结构是逐渐演变成下图这样的。



我们有五个团队：一个需求团队、三个功能开发团队和一个系统测试团队。还有一些人不属于以上团队，他们负责处理专门事务和协调工作，这些人是项目经理、项目管理专员、配置经理、电子学习专家、性能测试专家、开发经理、教练等。

三个功能开发团队基本上都是 Scrum 团队，也就是说，每个团队都在一起工作，多职能，自组织，能够开发和测试完整的功能。关于 Scrum 的更多信息，请参见 17.3 节。

需求分析团队实际上是一个虚拟团队，所以团队成员并没有坐在一起，而是分散坐开，这样保证项目团队所有人都能够就近与需求分析人员沟通。该团队中有三个子角色。

- 一部分分析人员隶属于具体功能开发团队，从开发到测试全程跟进该团队负责的功能，全程回答问题并澄清需求。
- 一部分分析人员关注项目的大局，并不隶属于具体功能开发团队。他们着眼于未来，定义概要功能区域。
- 其余成员较为灵活，根据当前的实际需要承担上述职责。

测试团队的结构与虚拟团队类似，也有相应的子角色。

- 一部分测试工程师隶属于具体功能开发团队，帮助开发团队测试并调试功能。
- 另一部分测试工程师则关注项目大局，集中对发布候选版进行系统的整体测试和集成测试。负责协调这项工作的人被戏称为*系统测试将军*。
- 其余人员较为灵活，根据实际需要承担上述角色。

过去，项目团队都按专业角色划分。我们原先有彼此独立的一个需求团队、一个测试团队、三个开发团队，开发团队都没有配备测试人员或分析人员。这种模式不利于团队扩展，因为随着越来越多的人加入项目，沟

通问题会越来越严重。每个团队都习惯于通过书面文件与其他团队沟通，而不是采用面对面直接交谈的方式，而且他们会把问题推给对方。每个团队都只顾完成自己团队的任务，而不关心整个产品。例如，需求分析人员写完需求文档并获得签发认可后，就认为自己的工作已经“完成”了，而不会全程跟踪该功能直到功能上线。

随着我们的开发团队开始转为 Scrum 式架构，各个团队具有了多职能特征，分析人员、测试人员和开发人员都坐在一起，团队之间的协作水平有了大幅提高。不过我们并没有彻底贯彻多职能团队的做法，还是有一部分分析人员和测试人员并不在功能开发团队里，这样他们就可以专注于项目大局，而不是单个功能。在这种模式下，团队很容易扩展，从而让我们既能够在短期着眼于功能，又可以长期着眼于产品，在二者之间找到了良好的平衡。

第3章

每天出席鸡尾酒会

如果你随便哪一天上午 10 点 15 分之前走进这个项目组，会感觉像进了鸡尾酒会现场！到处都是人，全都一小拨儿一小拨儿站在一起说话。



你会看到不同的白板前都有一小群人围成半圆形热烈交谈，还不停地把便利贴移来移去。你会看到有人中途离开一个小群体，加入另外一个小群体，辩论在继续，结论在形成。忽然，一小群人散开，其中一部分人转而加入另外一小群人继续交谈。有时候大厅中会形成新的人群，继续就一些未尽的话题交谈。

到10点15分时，鸡尾酒会散场，多数人则回到他们自己的座位上。

这种场面第一眼看上去似乎很混乱，实际上却非常有组织性。

3.1 第一拨：功能开发团队每日立会

首先是功能开发团队的每日立会。



其中两个团队9点30分开始开会，另一个团队9点15分开始开会（每个团队自行确定立会开始时间）。团队成员在任务板前大致围成半圆形，讨论他们当天要做什么以及需要解决什么问题。

萨利：我今天会研究一下那个可恨的内存泄漏问题。

杰夫：你可能需要先升级一下profiler工具。我上周就碰到几个问题跟这个工具有关。

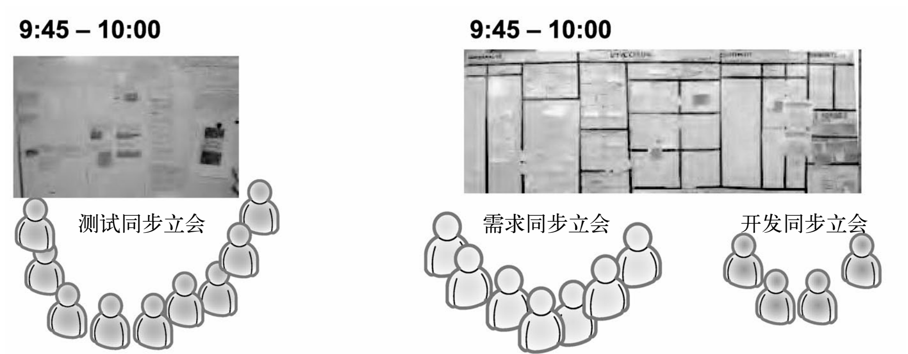
萨利：好的，谢谢提醒。如果有问题我会来找你。

有些团队遵循 Scrum 模式（来回答“我昨天做了什么工作”、“今天打

算做什么”和“遇到了什么麻烦”等问题），有些团队较为不拘形式。这种立会通常持续十到十五分钟，由团队主管（基本上等同于 Scrum 大师）主持。

3.2　第二拨：不同专业角色的同步立会

9 点 45 分，第二拨每日立会准时开始，各个专业角色（需求分析、测试、开发）的成员分别碰头，更新同步各个功能开发团队所做的工作。



所有测试人员都聚拢在测试状态板前，讨论如何最好地利用当天的时间。隶属于具体功能开发团队的测试人员刚刚跟各自所属的团队开完立会，所以他们可以分享各个团队的最新进展。

汤姆：我们今天需要把重点放在系统测试中的可用性问题上。如果谁能提供帮助，我们会非常感谢的。

丽萨：我大约一个小时后可以来帮你。我们团队今天会完成日志记录功能的测试。接下来就算我不在，他们应该也不会有任何问题。

与此同时，需求分析团队则在开他们自己的同步立会。刚刚跟功能开发团队开完立会的分析人员也来参加会议，他们有最新的信息可以向整个需求分析团队汇报。

吉姆：我们团队的人貌似对新出来的可用性指南很困惑。

约翰：我们团队也是！

玛丽亚：哦，难怪系统测试又成为瓶颈了。用户界面设计不一致，会让他们的测试工作很难进行。大家有什么建议？

吉姆：我们开个会讨论一下新出台的指南吧。

玛丽亚：好，回头我就在项目同步会议上提出来。我们今天找个合适的时间，至少要让每个团队都有一名开发人员和一名测试人员来参加。

与此同时，来自各个功能开发团队的主管则跟开发经理一起开功能开发同步立会。主管们刚刚从各自功能开发团队立会过来，有最新的信息跟大家分享。

杰夫：我们团队今天会完成日志记录功能的开发[指着墙上的一张卡片]。今天下午我们可能会开始做数据库迁移。

山姆：等等，这是不是说我们需要更新编译脚本啊？

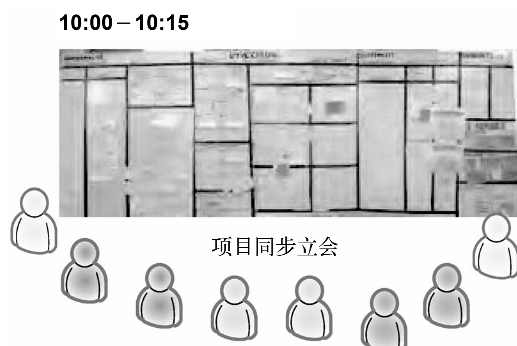
杰夫：对。不过很容易。如果你需要帮助，可以问丽萨。刚才团队立会的时候她说今天不忙。

测试团队围在测试状态板前开同步立会，需求分析团队和开发团队则分别围在项目看板（参见第 4 章“项目进度板”）前开同步立会。这三个立会同时进行，各自相距不过几米远，这样给人感觉有点吵闹有点混乱，但协作却非常高效。如果一个团队的人员需要另一个团队的信息，他们只需走几步到另外那个团队跟前，直接向他们提问即可。

有些人员（如项目经理和我）则在这几个立会之间走动，听取最新进展，了解有哪些高层问题需要解决。有时候我们不参与具体立会，有时候我们会应他们的要求加入讨论。

3.3 第三拨：项目同步立会

最后，10点整的时候，项目同步立会在项目看板前准时开始。



参加项目同步立会的人员被戏称为交叉团队，就像整个项目团队的一个横向剖面。我们这个项目里，就是每个专业派一个代表，每个功能开发团队派一个代表，再加上其他几个人，比如项目经理、配置经理和我自己。

我们在项目同步立会上审视整个项目的状况，主要关注从需求分析到投入生产的各个工序是否正常运转：每个团队今天在做什么？现在有什么问题阻塞了我们的流程？瓶颈在哪里？如何消除瓶颈？下一个瓶颈会出现在哪里？我们的发布计划进展是否顺利？有没有人不知道今天要做什么工作？

这不但能够让我们清楚地了解项目当前大势，还能够让我们迅速处理问题，特别是团队之间的协作问题。如果“我们”和“他们”每天都一起工作，那么“我们”和“他们”迟早都会变成“我们”。

就是这样。每天总共有七个立会，分三拨召开。每个立会的时间严格限定在15分钟内。每个立会都有必须出席会议的核心人员。每个立会又都对外公开，所以任何人想了解项目的信息或希望贡献意见，都可以随时加入任何一个立会。立会截止时间是上午10点15分。

如果每日立会上出现了15分钟内解决不了的重要问题，我们就跟相关人员单独召开后续会议来解决问题。有时项目同步立会一结束，相关人等就会围站在一起，讨论立会中提出的问题，有些最有意思、最有价值的讨论就是在这时候进行的。

这种每日立会的体制是我们慢慢形成的。刚开始实行“每日鸡尾酒会”时（顺便提一下，这只是我个人的说法，并不是项目中正式的叫法），我还有点担心大家会感觉会议太多。事实证明我多虑了。恰恰相反，团队成员们都认为这种立会很有价值，而且我发现大家都兴致勃勃，能够快速解决问题。

项目多数人员只需出席一个立会。个别人员需要出席两个立会，比如，功能开发团队主管需要出席团队立会和开发同步立会，隶属于功能开发团队的测试人员需要出席团队立会和测试同步立会，等等。这种方式把各条沟通渠道高效“挂钩”，保证了重要的知识、信息和决策在整个项目组内得以迅速传播。

原本需要通过创建文档和流程规则才能解决的很多问题，在这些早晨的立会中直接得到解决。比如我们会在立会中决定哪个团队开发哪个功能，又比如决定我们今天是花时间开发客户可见的功能，还是实施客户看不到的技术基础架构改进。各个团队只需在立会中对这些问题进行讨论就可以当场作出决策，而不用专门为这些问题制定新的策略规则。这就是大型项目保持敏捷、不为官僚程序所拖累的关键。

第4章

项目进度板

项目进度板是项目的沟通枢纽。我们的项目进度板是一块几米长的白板，显示项目所有关键功能从需求、开发、系统测试到上线整个流程的最新状态。



如果你对看板有所了解，就会一眼看出这就是看板系统，也就是说我们在跟踪从创意到功能上线的过程中的价值流，并且在整个流程中的每一步都会控制在制品工作量。有关看板的更多内容，请参见 17.5 节。

下面概要介绍我们的看板中每一栏所代表的含义。

最左边的一栏是收集到的创意。这些创意都是概要功能区域。每个功能区域都写在一张综述卡上。例如，写有“没收”字样的卡片代表与没收嫌疑人物品相关的所有功能。



综述卡迟早会被拉入第二栏（正在分析），我们会在这里对其进行分析，并从功能角度将其分割成若干个用户故事。用户故事都会写在第三栏中的功能卡上。第三栏大致与 Scrum 产品需求清单对应，不同的是功能卡并不严格排序。多数功能卡都以用户故事的格式写就：“作为 X，我需要 Y，以便能够 Z。”例如：“作为调查员，搜索地址的时候我需要按区域过滤，以便能够快速查找到地址。”

如果某个综述已被分析过（即已细分成若干功能），相应的综述卡就会被丢弃，代之以第三栏若干更为详细的功能卡。因此，综述卡绝对不会流出第二栏，功能卡则诞生于第三栏。

功能卡是白板上的主要“货币单位”。

我们会挑出最重要的十个功能卡放入“下十个功能”栏中。通常我们会在每两周一次的会议上挑选功能。这个会议大致对应于 Scrum 的 Sprint 规划会议（其实我们就是这样称呼的）。至于如何挑选出十个功能的具体做法，请参见第 13 章。

接下来，三个功能开发团队会根据各自的情况，把功能卡从“下十个功能”栏拉入他们自己的“正在开发”栏，完成功能开发和测试工作后，再拉入“等待系统测试”栏。

测试团队定期清理“等待系统测试”栏，把功能卡拉入“正在进行系统测试”栏（并在版本控制系统中创建一个相应的系统测试分支，详细内容请参见第 14 章）。一旦系统测试完成，测试团队就会将其放入验收测试环境，把功能卡拉入“等待验收测试”栏，然后返回，对新开发完成的功能开始新一轮的系统测试。这种做法是组织文化上的一大转变，即从“版本发布周期最后阶段才做大型系统测试”转变到“（分批）持续进行系统测试”。

每隔两个月左右，都会有一批真正的用户过来，花两三天的时间做验收测试（基本上只是试用一下系统并给出反馈意见），这时候我们就会把功能卡挪入“正在进行验收测试”栏。等到验收测试完成，发现并修复了最后的 Bug 以后，功能卡就会挪入“等待投入使用”栏。之后不久（产品发布后），功能卡会被挪入最后的“投入使用”栏。功能卡在这一栏会停留几周（显示我们有些成果正式上线了），不过之后就会被清理掉，以便给新来的功能卡腾出位置。

对随便扫一眼看板的人而言，我们的体系看上去很像瀑布式开发模式：需求分析→开发→系统测试→验收测试→核准上线。其实还是有很大区别的。在瀑布式开发模式中，所有需求分析在开发工作开始之前就已全部完成，所有开发工作则在测试工作开始前就全部完成。而在看板系统中，这些不同的阶段都是并行的。第一组功能进入用户验收测试阶段时，第二组功能还在系统测试阶段，第三组功能正处于开发阶段，第四组功能则处于需求分析和用户故事细分阶段。这是一个从创意到产品上线连续不断的价值流。

不过，我得说是半连续的价值流。我们的项目或多或少是一个从创意到“等待验收测试”阶段的连续价值流。大致每隔两个月才会核准新功能上线，而且是跟验收测试一起完成，所以新功能会在“等待验收测试”阶

段停留几周。我希望未来会有所改进，不过实践证明这样做问题不大。因为在开发阶段会有现场用户给出反馈意见，所以我们发现，一个功能到了“等待验收测试”阶段时，就基本上已和我们的预期相符，不会再出现什么严重问题了。

运用看板，发现 Scrum 的价值

这似乎已成为一个普遍模式：我注意到很多看板团队都会逐渐发现（有时候是重新发现）Scrum 很多做法的价值。实际上，有些看板团队之所以开始采用看板，是因为不喜欢 Scrum，可后来他们却发现 Scrum 其实相当好用，而且出现的问题并不是由 Scrum 引起的，恰恰是因 Scrum 而得以曝光。他们的真正问题是在运用 Scrum 时过于“照本宣科”，而没有根据自身的具体情况灵活运用。

更多内容请参见我的另外一本书 *Kanban and Scrum: Making the Most of Both*[KS09]。

4.1 我们的节奏

节奏（cadence）是以固定周期反复发生的事情，它构成项目的节律或心跳。我们的项目节奏大致如下图所示。



- ❑ 每两周回顾总结一次（有些团队每周一次），寻找改进流程的方法。
- ❑ （大约）每两周开一次规划会议，决定接下来要专注于哪些功能。

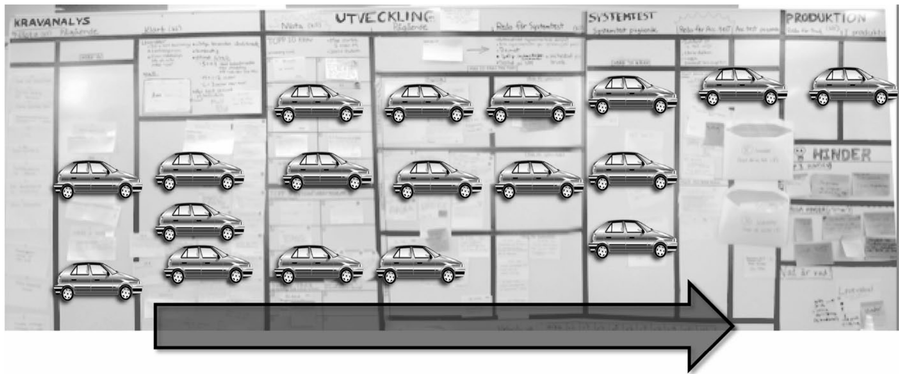
- 随着功能的开发与测试，演示与系统测试持续进行。
- 大概在每两个月的阶段末期核准上线。

我们始终在不断改进，越来越朝着 Scrum 式的模式发展。起初，回顾总结会的频率是规划会议的两倍；现在这两个会则是每两周才开一次，一个早一天，一个晚一天。演示与回顾现在是持续进行，不过我们在考虑每两周增加一次产品的全局演示/回顾。你猜怎么着？把回顾总结、规划和演示都以同样的节奏进行，基本上就是 Scrum 迭代（sprint）的定义。

向 Scrum 模式转变并非我们有意为之，这不过是为了解决现实问题所做的一系列流程改进工作带来的必然结果而已。

4.2　如何处理紧急问题和障碍

将看板比作交通系统非常形象。可以把一块任务板视作多条道路，每张卡代表一辆需要从左边驶向右边的车。



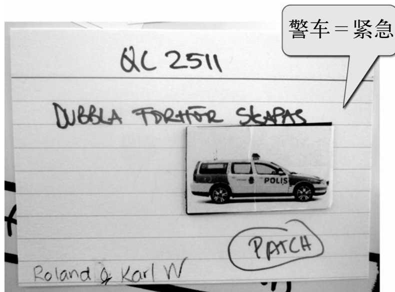
我们需要优化工序流，所以不能把任务板塞满。大家都知道路上塞满了车是什么结果，那时整个交通系统会瘫痪。



我们需要空间或裕度来应变，让工序流动更快。



系统有了裕度不仅能让流程快速流动起来，还便于处理紧急情况。我们在任务板上用警车磁贴（本来就是警局项目嘛）标出需要特别快速处理的紧急事项。



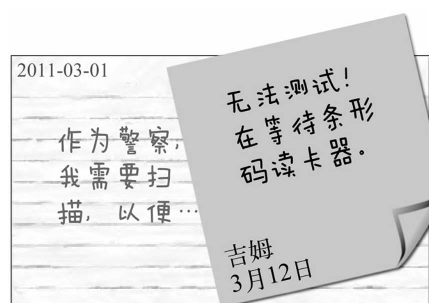
我们还用粉红色便利贴来标示障碍（“路障”）。



如果某个功能进展停滞（例如，缺乏测试该功能必需的第三方工具），我们就在相应的功能卡上贴张粉红色便利贴，写清问题和停滞开始日期。任务板右边还有一小块区域是“前三个障碍”，显示与具体功能不直接相关的普遍问题（如编译环境出问题了）。

我们在每日立会上着重移除这些障碍。就像交通系统一样，若障碍长时间得不到解决，就会对整个系统产生影响。而且，瓶颈路段会让全路段所有车辆慢下来，所以我们把所有精力都集中在解决这些瓶颈问题上。

下面是一个在每日项目同步立会上解决障碍的例子。



艾瑞克：那么吉姆，停滞的这项现在是什么状态？

吉姆：还是没有条形码读卡器，本来上周就应该送到的。我也不知道什么时候能拿到，所以还没法测试。

艾瑞克：嗯，我们只能干等吗？还能有什么其他办法？

崔西：我做的上一个项目用到过条形码读卡器，说不定还能找到一个。

吉姆： 有可能型号不是我们要的，不过我可以试试。也许能成。

艾瑞克：好，同时我会跟客户反映一下这个问题，然后再给供应商加点压力。你还有什么其他需要吗？

吉姆：没有了，我想我们今天能做的就是这些了。

如果问题第二天还是没解决，粉红色便利贴就会留在那里，提醒大家需要跟进这个问题。便利贴上的日期表明问题存在了多久，落款则表明谁在负责解决问题（这样我们就知道这个问题该找谁）。

艾瑞克：我看到条形码问题还在那里……

吉姆：对。崔西和我试过她的读卡器，型号不对，所以我们还是没法测试。

艾瑞克：真糟糕。对了，我昨天问过供应商，不过也没得到明确答复。我还跟客户提过这个问题。不过他们说条形码功能在这次的发布版本中没那么重要，既然有问题，我们可以先跳过这个功能。

吉姆：太好了！那我可以开始测下一个功能了。

艾瑞克：我也会找个新的供应商，这样等以后再测试这个功能的时候我们就做好准备了。

项目进度板或许是项目沟通最重要的工具，让大家对项目的整体状态一目了然，而且还能实时显示工序流动状态和瓶颈问题。

问题是：这样一块任务板如何在60多人的项目中发挥作用？请看下一章“扩展任务看板”。

第5章

扩展任务看板

项目的开发速度很大程度上取决于团队成员对项目当前状态的熟知程度。如果人人都清楚项目的当前进展和未来走向，就比较容易同心协力向目标挺进。

随着项目成员增加到60多人，这就成了个问题。每个团队都有自己的任务板，显示团队自己的工作进展，包括正在开发测试哪些功能，谁负责某个功能的某个具体任务等等。可是，我们却不清楚项目的整体状况。项目整体进展如何？现在的瓶颈在哪里？接下来会有哪些新功能？到这次版本发布的时候，哪些功能可以按时完成？

于是，我们创建了项目进度板，用来显示项目所有功能从需求、开发、系统测试到核准上线的最新状态，以便让我们对项目的整体情况一目了然。

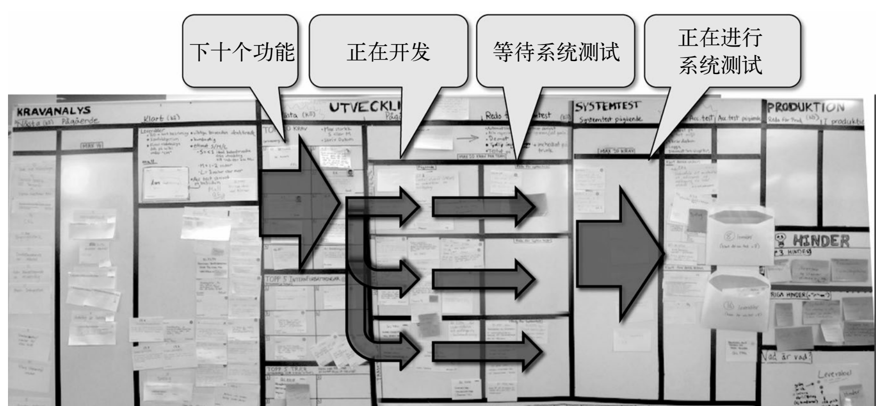


这块任务板对组织文化有很大的影响。现在我们能够看到了！而且我们所有人看到的都是同样的项目状态！

团队之间的协作水平得到显著提高，因为每个团队都能看到他们的工作会如何影响（有时候是扰乱）功能上线的整体流程。

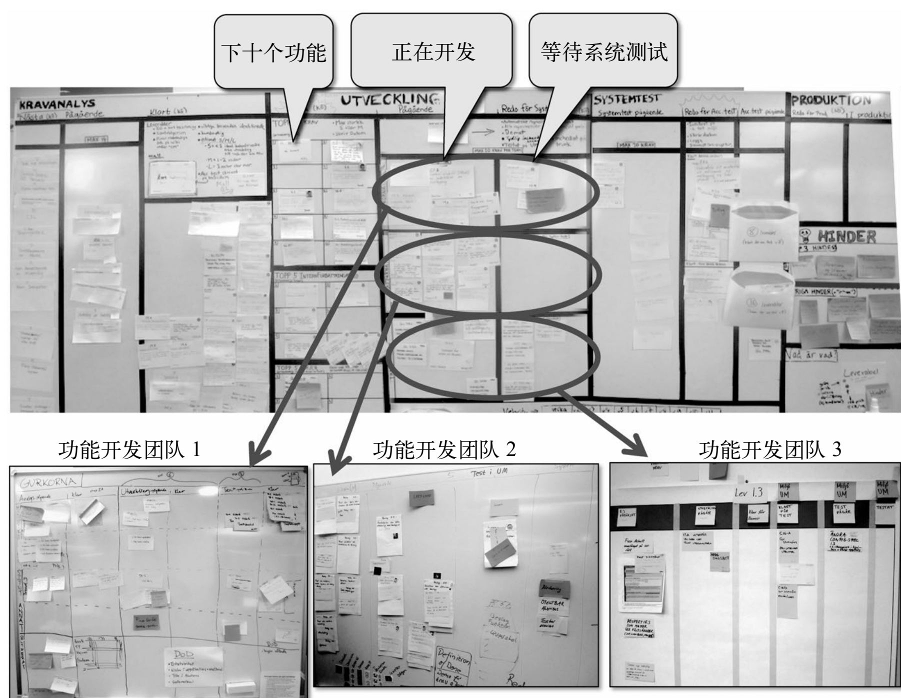
不过，我们并不打算撤掉各个团队的任务板，因为团队任务板还是很棒的，能将每个团队正在开展的日常具体任务可视化，有助于保持团队成员信息同步。而且我们也不想把只跟具体团队有关的所有详细信息都放到大的项目进度板上，否则会太混乱，反而看不清项目全局。因此，我们决定保留两个层次的任务板——一块大家共享的项目进度板加三块团队任务板。

项目进度板上的开发栏从上到下分为三行，每个功能开发团队一行。



“下十个功能”栏中的每个功能卡都会流入三个功能开发团队之中的一个团队。等到功能开发团队完成了一个功能的开发和测试工作，这个功能就会流入“等待系统测试”栏。等到系统测试团队完成其上一轮系统测试工作后，就会将新功能卡从功能开发团队的“等待系统测试”栏拉入统一的“正在进行系统测试”栏，并开始新一轮的系统测试工作。至于我们是如何开展测试的，请参见 9.1 节。

功能开发团队每次从“下十个功能”栏拉一张功能卡到“正在开发”栏，都会复制一张卡，贴到他们自己的团队任务板上去。



然后，他们把工作细分成具体任务，写在便利贴上，然后贴到功能栏上。通常这是在分析立会中顺便完成的。需求分析人员、测试人员和开发人员在分析立会上一起协作，勾勒出该功能的大致设计，然后确定主要任务。每项任务都会写得简明扼要，通常用动宾结构，例如，“写 GUI 代码”、“建数据库表”或“设计协议”。

所以，项目进度板上贴的是功能卡，而功能开发团队的看板上则贴有他们负责开发的具体功能和相关细分任务。想象一下你“双击”了项目进度板上的一张功能卡，然后就可以“放大”相应的团队任务板，查看该功能都涉及哪些具体任务以及谁负责哪项任务。

多数功能开发团队还用头像磁贴来表示谁在负责哪项任务。而头像则能说明相关人员的个性……



如图所示，每个团队都有自己的任务板布局。我们没有规定标准的任务板布局，而是让各个团队自行寻找最适合他们需求的任务板结构。多数团队在他们的任务板上都标有同时进行的最多工作数（即 wip 高限，参见第 11 章）、“完成”的定义（参见第 7 章）和头像磁贴。

采用这么两个层次的看板体系，尽管最初有人担心如何保持信息完全同步，但实践表明这种体系非常有效。很明显，任务板已成为项目的核心；想要了解工作状态或解决问题，大家都会自然而然聚集到各自团队的任务板前。团队成员主要关注自己团队的任务板，团队主管和经理则同时也关注项目整体进度板。

随着时间的推移，越来越多的团队成员开始对项目整体进度板产生兴趣，这是个好现象，表明大家开始关注项目全局，而不仅仅是自己手头的工作。

不过，如果真心需要大家齐心协力向目标前进，我们还需要明确定义一个总体目标。

第 6 章

跟踪总体目标

如果人人都清楚总体目标是什么，就会更关注总体目标。

我知道这听上去不言自明。但在我服务过的很多组织中，管理人员都以为项目成员清楚他们的总体目标。可事实却表明，在被问到总体目标是什么时，每个人都会有自己不同的答案。

我们的项目总体目标通常就贴在项目进度板上。例如，2011 年第一季度时，我们的目标是“4 月 5 日交付没有重大缺陷、可以向全国范围发布的版本”。实现这一目标之前，还有一个里程碑，即 3 月 14 日向两个新增地区发布。



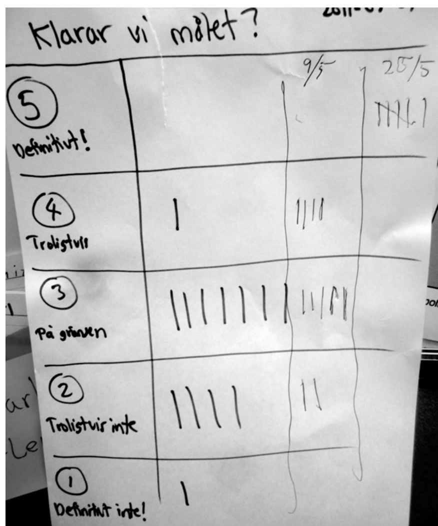
实现一个目标后，我们就为下一个发布版本写下新的目标宣言。目标宣言就是指路明灯。有时我们需要作出艰难的取舍，而明确的总体目标有

助于项目所有人员保持同步，明白对于下一个发布版本而言什么最重要。

我们每周或每两周都会做一次现实检查。通常项目经理会在项目同步立会上问大家：“你认为我们能实现这个目标吗？”然后每个人写下 1 到 5 之间的一个数字（有时候我们直接举手指）。

- 5 = 绝对能
- 4 = 有可能
- 3 = 有点悬
- 2 = 不太可能
- 1 = 拉倒吧

下面是一个例子。



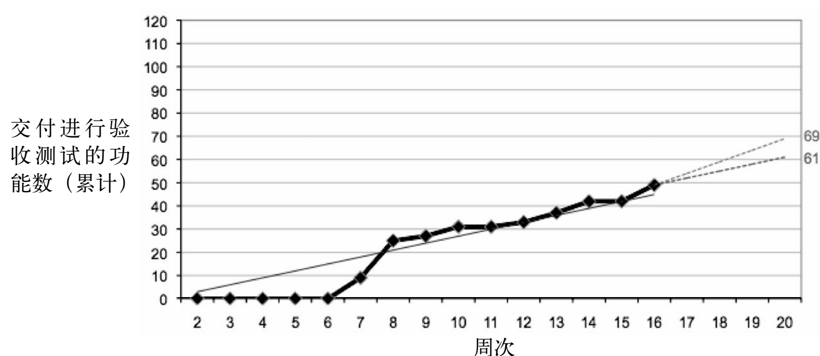
这张图上是三轮投票的结果。第一周投票的时候（最左边一栏），大家对实现目标信心不足；第二周信心有所上升；第三周就信心满满，全都是 5 了！

只要投票结果中有 2 和 1，我们就会重新评估目标，并讨论需要做什么改变来提振信心。下面是一些可以采取的典型行动。

- 消除障碍（“买台新的编译服务器，换掉有故障的那台！”）
- 帮助解决瓶颈问题（“今天我们所有人都做测试！”）
- 缩小范围（“如果把功能 X 从这次的发布版本中去掉，就还有可能实现目标！”）
- 调整目标（“这个目标已经不现实了。我们来定个能真正实现的新目标！”）
- 更加努力工作（“周日谁能来加班？”）

前四条中的任何一条都比第五条管用，因为问题的根本原因不是我们工作不够努力。实际上，有时候问题的根本原因是我们工作太努力了，以至于没时间思考。

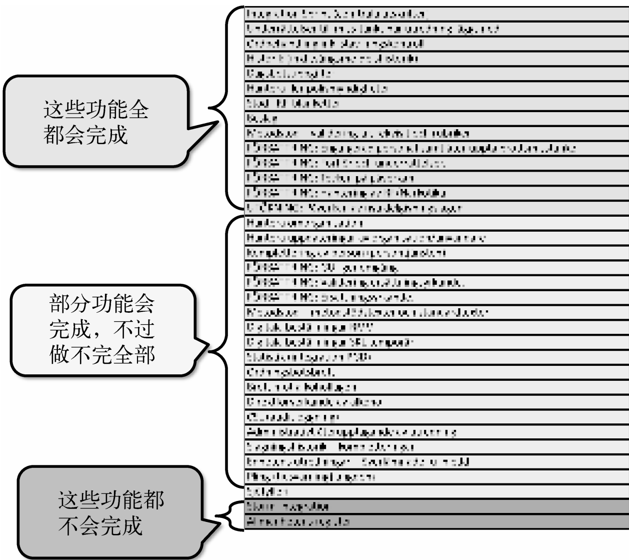
投票基本上凭直觉，不过从某种程度上而言，还取决于大家都可以看得到的信息，包括任务板上的功能卡，周期时间和速率等度量指标（参见第 12 章），以及各类图表，如下图所示的功能燃尽图。



横轴代表周，纵轴代表截止某周已经完成的功能数量。这张燃尽图形象显示了随着时间推移系统开发的完成情况。

右边伸出的两条虚线是趋势线，分别代表对下一次版本发布时能够按时完成的功能数量的乐观预测和悲观预测。当然，我们还只是在猜测，不过我们的猜测是基于经验数据的，而不是胡乱猜测。

有了这一数据，我们就可以制订出简单现实的版本发布计划，不会试图隐藏任何不确定性。



随着发布时间越来越近，我们会越来越清楚哪些功能可以做完，哪些做不完，所以不确定性（如上图中段所示）就会降低。

顺便提一下，我们并没有使用任何花哨的项目管理工具来生成这些图表。我们用的就是简单的电子表格。

无论如何，持续做这种现实检查是检测和避免死亡行军（人人都知道会失败却仍然继续做下去的项目）行之有效的简单技术。如果大家能够对自己认同的目标形成共识，就会积极促进自我组织和协调。反之，如果大家不了解目标，或者对实现目标没有信心，就会不由自主地将自己与业务目标脱离，只关心个人的目标，比如“只管开心写代码”或“做完分内的事就回家”。

无论你参与的是什么样的项目，我都强烈建议你们明确定义目标并定期做现实检查。这种做法成本很低，但收益却非常高！

第 7 章

定义“可供”与“完成”

明确进度看板上的各栏的含义非常重要，对大型项目而言尤为如此，因为参与的人越多，混淆的风险就越高，随之造成的代价也就越高。

在项目进度看板大多数栏的顶部，我们都用蓝色彩笔写下相应栏的“完成”的定义（即针对下一栏的“可供”的定义）。对我们而言，最重要的两个定义是“可供开发”和“可供系统测试”，因为这两处是我们以前出问题最多的地方。

“可供开发的定义”

“可供系统测试”的定义

7.1　可供开发

“可供开发”栏实质上表示“我们这里有一堆已细分好任务、估算完工作量和澄清了范围的功能，不过我们还没决定要开发哪几个功能，以及按什么顺序开发”。所以，这与 Scrum 产品需求清单大致对应。一个功能要进入可供开发状态，就必须具备以下特点。

- 必须有一个 ID。如果有与该功能相关联的用例规范或其他文档，就可以通过 ID 来查询这些相关信息。点击项目 wiki 文档站点上的相应 ID 就能够访问这些文档。
- 必须有一个联系人。联系人通常是需求分析师，拥有该功能相关领域的丰富知识。
- 必须对客户有价值。把综述分割成可交付的用户故事时，我们需要确保未丢失客户价值。需求分析师对此有最终发言权。
- 必须经由团队估算。通常由一名测试人员、一名开发人员和一名需求分析人员组成的小组利用规划扑克进行估算（参见第 19 章）。我们用 T 恤衫尺寸（大、中、小）来表示。这种估算是估算工作量的多少，而不是估算时间长短。不过为了简化估算过程，我们遵循下面这些准则。
    - 小指的是“在没有任何意外的状况下，大约花不到一周的时间就能够到达‘可供验收测试’阶段。”所谓没有意外状况是指我们找对了人，在没有干扰的情况下，专门负责这项功能。
    - 中指的是一到两周的时间（同样，在没有任何意外的状况下）。
    - 大指的是超过两周时间。大一些的功能可供开发前需要进行进一步细分。
- 必须将验收测试情景写在功能卡背面，即描述典型测试情景的具体步骤，示例如下：“乔警官登录，查询 235 号案件，然后关闭该案件。

然后他再次查询235号案件，看到该案件已关闭。”

7.2 可供系统测试

“可供系统测试”是指功能开发团队已完成他们能够想到的所有工作来保证某项功能正常，而且没有任何重大缺陷。不过，交付系统测试之前，他们只专注于测试该功能本身，而不是整个产品。

长久以来，系统测试都是一大瓶颈，其中一大原因就是在系统测试阶段会发现大量不必要的缺陷。“不必要的缺陷”是指远在开始系统测试之前，在功能测试阶段就应该发现的缺陷。因此，我们的可供系统测试的定义就把质量门槛提得较高，便于尽早发现非常麻烦的错误，同时要让功能开发团队也对质量负责，在将功能交付给系统测试团队并转向开发下一个功能之前，允许他们花费必要的时间确保一项功能的质量。

因此，我们的可供系统测试定义如下。

- 验收测试自动化：端对端功能层次的验收测试或集成测试已自动化。我们以前用Selenium测试工具（该工具直接从Web界面运行测试脚本），不过后来转用 Concordion 自动化测试框架了。对于我们谜一般的Ajax Web界面而言，Selenium工具过于脆弱。随着我们转向实例化需求驱动开发（Specification By Example），Concordion自动化测试框架更符合我们的需要。
- 通过回归测试：对现有功能运行的所有自动化测试都要通过。有时一个新功能会破坏现有功能，所以我们必须要保证定期运行所有回归测试。
- 已演示过：开发团队已向团队其他成员、现场用户、需求分析人员、系统测试人员和可用性专家等人演示过该功能。演示的目的是尽早发现可用性方面的问题，这样就不会等到系统测试阶段甚至用户验

收测试阶段才发现问题。

- 清楚填写签入备注：签入该功能相关代码时，应当用该功能ID来标记签入备注，并填写容易理解的备注，说明都做过什么。这将保证最低限度的可追溯性（大型项目的可追溯性总是糟糕得一塌糊涂）。
- 在开发环境中测试过：每个团队都有专用的测试环境，而且测试过该功能，免得开发人员说“在我机器上运行完全没问题”之类的话。
- 已与代码主干合并：该功能代码应当已经合并到代码主干，而且任何代码合并冲突都已解决。这是我们所使用的稳定主干模型的基础（请参见第14章）。

7.3 两个定义如何提升团队协作

可供开发和可供系统测试这两个定义显著提升了团队之间的协作。我做了一个简短的调查，想了解一下大家对流程变更的看法，调查结果表明团队协作改善显著。

过去我们刚开始采用看板时，各个团队基本上都只关注项目进度板上“自己”那一部分内容。需求分析人员只看项目进度板左边那一块，认为写好某个功能的需求文档后自己的工作就做完了。开发人员只看中间那一块，而测试人员只看右边那一块。测试人员不参与需求文档的撰写，所以等到一项功能需要进行测试时，他们甚至都不清楚这项功能该怎么用。于是大家花了很多时间争论需求文档应该详细到什么程度。

这些都是老习惯了。有了项目进度板后，大家都看到了问题，而发现问题正是解决问题的第一步，也是最关键的一步！

经过几周的时间，大家对定义形成共识之后，团队协作问题开始慢慢消失（嗯，至少是大幅减少）。只有不同角色的人员一起合作，对功能进行估算，然后细分成很小却又不会失去客户价值的可交付单位，并就验收测

试达成一致意见，可供开发的状态才算真正实现。

同样，只有不同角色的人员一起合作，从功能层次进行测试（包括自动化测试和手动探索性测试），来确定该功能可以放行，可供系统测试的状态才算真正实现。

正是这种对持续协作的明确需求，才让测试团队和需求分析团队同意向各个开发团队“借出”他们的专业人员，从而让每个开发团队真正具有了跨职能（而且更为高效）的特点！

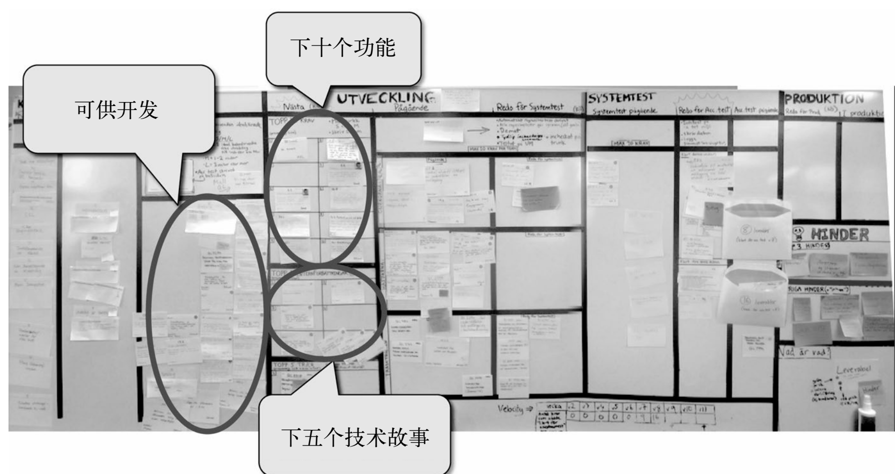
总之，在任何的看板系统中，在每一栏的最上面写下就绪（可供）的定义都是很简单却又行之有效的一个技巧。

第8章

处理技术故事

技术故事是一些我们需要完成但客户并不感兴趣的事情,比如升级数据库、清理不用的代码、重构糟糕的设计或对原有功能实现测试自动化等。我们通常称之为内部改进。不过，细想一下，既然我们说的是跟产品相关的技术工作，而不是流程改进本身，所以用技术故事这个术语似乎更为合适。

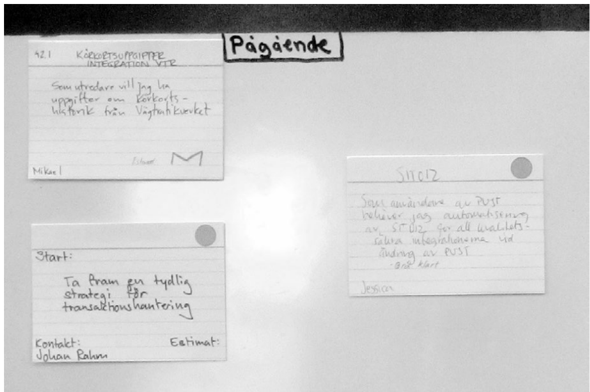
技术故事诞生于项目进度板的“等待开发”栏，通过“下五个技术故事”栏（就在“下十个功能”栏下方）流入开发阶段。这两栏是平行的，都是开发栏的输入队列。



如图所示，“可供开发”栏中的卡片数量相当多，囊括了功能卡和技术故事卡。我们并没有浪费时间去给所有这些卡片排出优先级，而是不断地根据当前任务的紧急程度确定下十个功能和下五个技术故事。这样也就预留出了足够的工作，保证开发团队不会无事可做。

当开发团队有空接手新的工作内容时，他们要么从“下十个功能”栏拉一张功能卡过去，要么从“下五个技术故事”栏拉一张技术故事卡过去。如何在这两者之间保持平衡并没有一成不变的规则，需要在每日立会中不断讨论和调整。

技术故事卡右上角上有一个圆点（一般涂成绿色），以跟功能卡区分。即使被拉入开发阶段后，我们也能凭这个圆点将两者区分开来，因此项目进度看板就能显示出我们在功能和技术故事上是如何分配时间的。



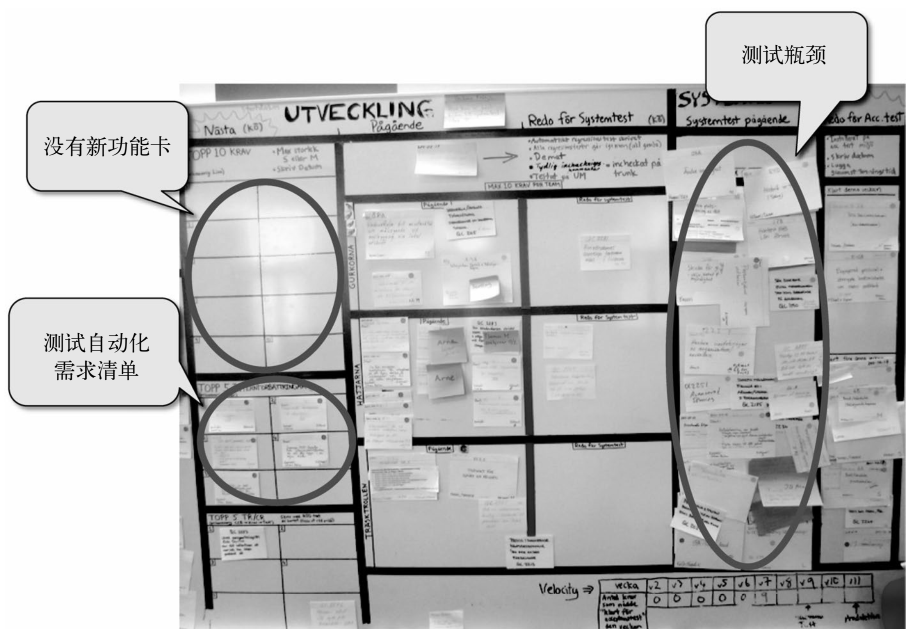
通常功能更重要，不过也有一些例外情况，让我们需要在一段时间内花费大部分时间和精力专注于技术故事，比如下面的例子。

8.1 示例 1：系统测试瓶颈

系统测试一度成为很突出的大瓶颈，所以当时已经完全没有意义再开发新功能，继续给瓶颈增加负荷。发现这个问题后，开发人员就开始专注

于实施可减轻系统测试负荷的技术故事，主要是测试自动化的内容。实际上，测试经理当时的任务是创建测试自动化需求清单并进行排序，然后通过“下五个技术故事”栏提供给开发人员。测试人员于是就变成客户了！

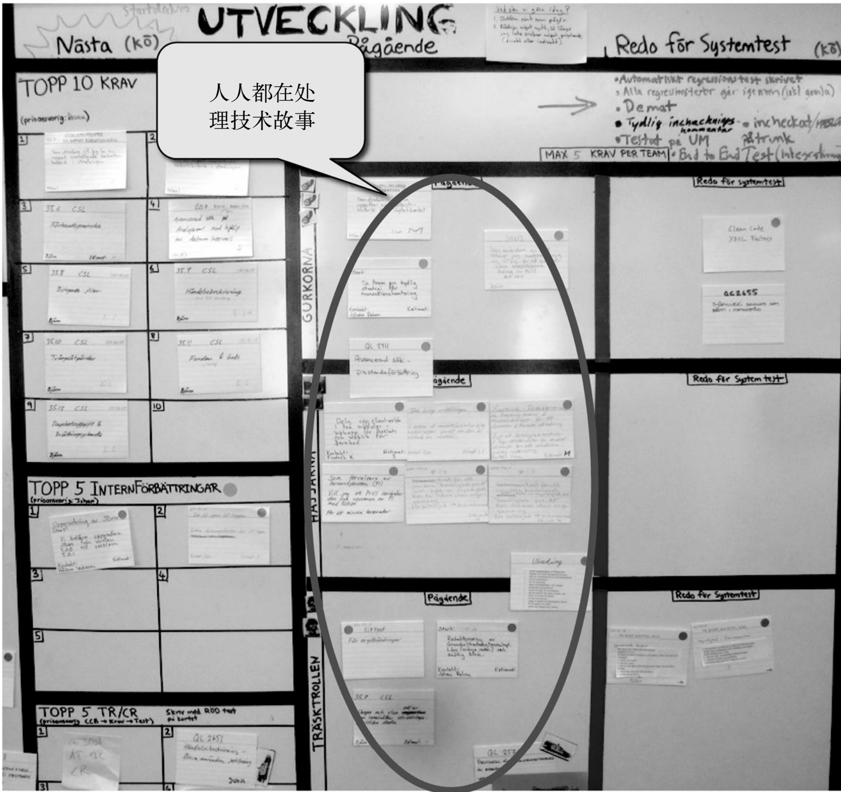
有关这项技术的更多内容，请参见第 18 章。



8.2 示例 2：版本发布前一天

在主要版本发布的前一天，团队都希望先将该版本送出门，然后再开始接手新一轮的新功能。于是他们会专注于最后的 Bug 修复。如果当时没有任何 Bug 需要修复，他们就会处理技术故事——通常是些我们一直以来都需要完成但没时间做的工作，比如清理不用的代码、完成重构、学习新工具的使用等等。

如图所示，进度看板显示我们有很多技术故事（带有圆点）正在进行。



8.3 示例 3：7 米长的类

用下面这种方式将技术故事商业案例化一定很酷。我们的代码库中有一个类有点失控，需要进行彻底重构，但有的人不同意在这上面花时间。于是，一位团队主管把整个类打印出来，铺在了会议桌上！竟然有整整 7 米长！

看着如此冗长的内容，大家都心知肚明，必须要用一个技术故事来马上修复这个类！无需任何讨论。这个例子同时也说明，一直以来对设计关注不足而仓促行事，后果会有多严重。

作者评语……

我还从来没见过这种规模的项目到版本发布的时候居然没出什么大事！真让人失望……

版本发布前一天惯常的恐慌和忙碌以及连夜加班现象都哪里去了？发布后接踵而来的狂轰滥炸般的支持问题和慌不择路的热修复补丁都哪里去了？我是在最重要的版本发布（全国范围的版本发布，也是整个项目的核心）后第二天加入项目的，几乎没看到任何发生过重大事件的迹象。

所有这一切都得益于现场用户和试点版本，在版本发布之前就已经得到充分的实战演练。当然，之前的试点版本的确有一些问题——但这也正是我们需要发布试点版本的原因，对吧？

无论如何，要记得庆祝版本发布成功——即便是你已经习惯这么平淡的发布，不觉得有多兴奋也要庆祝。

顺着这一主题，我们开始天马行空般揣测今后的开发工作。我们按代码米来估算功能怎么样？每天按代码米来衡量开发工作的速率如何？我们甚至可以把理想的代码米（就是如果保持代码十分整洁，代码应该有多长）与实际代码米分开。把这两项相减，你得到的结果就是欠了多少技术债——以米为单位！我们甚至还可以在地板上划条线来表示还欠多少技术债（“喂，看看吧，我们还欠23米的债!”），没准还可以考虑用代码英里来衡量……

嗯，好吧，我还是闭嘴为妙。

不过，既然已经提到了代码质量，我们就来谈一谈Bug吧。



第9章

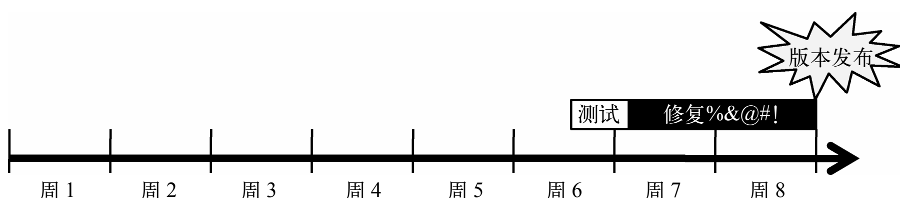
处理 Bug

转向看板之前，我们曾以传统方式处理 Bug。测试人员在开发周期末尾的系统测试时找到 Bug，然后记录到 Bug 跟踪系统中。Bug 跟踪系统中有成百上千个 Bug，变更控制委员会则每周碰头，对 Bug 进行分析并排序，然后分配给开发人员。对所有参与的人来说，这个流程都是既枯燥又低效的（不幸，所有人都牵涉进来了）。

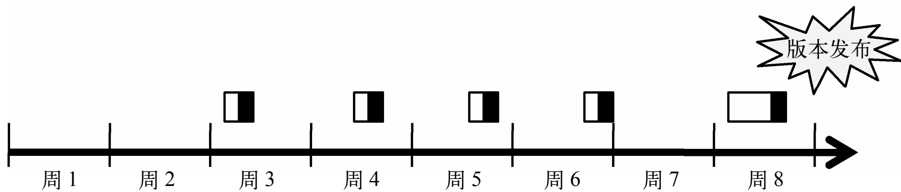
9.1 持续系统测试

看板体系帮助我们看清了一点，那就是我们需要持续（嗯，至少是定期）做系统测试，而不是全部堆到最后一刻。测试团队起初很抵触，因为系统测试很花时间，而且给人感觉在版本发布周期内做好几次测试很没有效率。其实这是个假象。最后才做系统测试看上去似乎更有效率，但如果把修复 Bug 的时间也算进去，就相当没有效率了。

下图是在一个为期两个月的版本发布周期内，最后才做系统测试的情景。

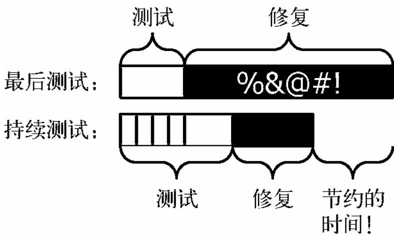


如果多做几次系统测试，则会是这个样子。



不到最后就无法测试整个系统，因为系统还没完成。不过，在已完成的功能基础上，我们仍然可以提前运行一部分系统测试。而且到最后还可以再运行一次完整的系统测试。最后的系统测试可能花的时间跟以前一样多，但修复 Bug 的时间大幅减少——而这才是真正花费时间和精力的大头！修复 Bug 的时间减少是因为我们已经提前发现并修复了很多 Bug，而最后测试时发现的通常都是新出现的 Bug，因此更容易发现并修复。提早发现 Bug 也加速了我们的学习过程。

那么，让我们把这两张图放在一起，对比两种情况下的测试和 Bug 修复时间。



这张图非常重要。再仔细看一看，尤其是测试人员。对，第二种情景下测试时间的确增加了，但总共需要的时间却减少了！

当然，还有一个关键要素，就是自动化测试。我们不可能把所有测试都自动化，不过既然要一次次反复进行系统测试，我们就需要尽可能实现测试自动化！

9.2 立马修复 Bug！

现在，测试人员发现一个 Bug 后，并不是直接记录到 Bug 跟踪系统中，而是写在粉红色便利贴上（像记录其他障碍一样），然后去找开发人员。多数情况下，他们大致知道该去找谁，因为每个开发团队都有一位专职测试人员。否则，他们会找开发团队主管，问清谁是能够修复 Bug 的合适人选（通常是写那段代码的开发人员）。

接下来，开发人员和测试人员坐在一起现场修复 Bug，或者开发人员自己修复 Bug，然后马上再去找测试人员。这样做的特点是，没有移交、没有延迟、不需要通过 Bug 跟踪系统来沟通。这种做法比较有效率，原因如下。

- 早发现并修复 Bug 比晚发现和修复 Bug 更有效率。
- 面对面交谈比书面沟通更有效率（沟通更充分）。
- 大家都会学到更多知识，开发人员和测试人员能够相互了解对方的工作内容。
- 不需要浪费很多时间去管理冗长的老 Bug 列表。



小乔爱问：
你真的会忽略那些没排进前 30 名的 Bug 吗？

嗯，好吧，有时我们会记到 Bug 跟踪系统里，不过会把状态设成延期（Deferred），就等于在说“对，我们已经知道这个问题了，不过可能现在没时间修复”。我们这样做的主要原因是，测试人员好不容易发现一个 Bug 却被我们直接忽略掉的话，会让他们感觉很受挫。即便有可能永远都不会修复这个 Bug，只要记下来，大多数人都会或多或少从心理上感觉安慰一点。此外，这样做有可能在做数据挖掘的时候有用，例如，需要统计看看系统的哪个部分 Bug 最多。

不过，延期的 Bug 基本上不在前 30 个 Bug 之列，所以延期实际上就等于把不需要却又不忍心丢弃的物品暂存在地下室。

有时候，一个 Bug 不够重大，不需要马上修复——例如，如果它只是会给用户带来些微的不快，而实施其他功能却更为重要。在这种情况下，对，测试人员会把它记到 Bug 跟踪系统中。当然，除非系统已经满了，无法记录。

你说什么，满了？Bug 跟踪系统怎么可能会满？

9.3 为何要限定 Bug 跟踪系统中的 Bug 数量

转向看板前，我们的 Bug 跟踪系统记录有成百上千个 Bug。现在我们硬性规定只能记 30 个。

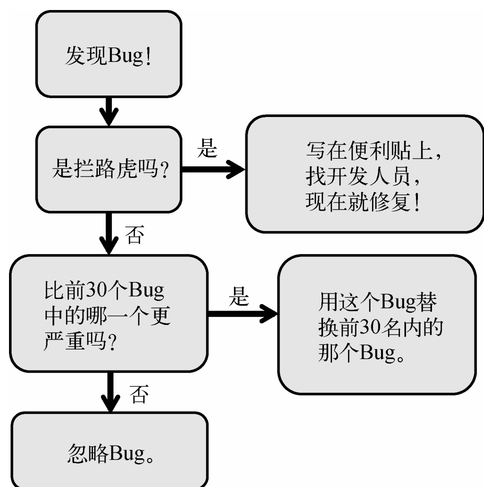
发现一个 Bug 后，第一个问题是"这是不是个拦路虎？"拦路虎的意思是"这项功能因为这个 Bug 而不能发布"或"修复这个 Bug 比开发新功能更重要"。我们把这样的 Bug 写在粉红色便利贴上，像对待其他障碍一样马上就修复，而不把它排进等待队列中。

不过，如果这个 Bug 算不上拦路虎，我们就需要作出决策："它比 Bug 跟踪系统中那 30 个里的哪个 Bug 更严重吗？"如果是这样，就把不重要的那个 Bug 从前 30 名列表中拿掉，给新 Bug 腾出位置。如果不是，那我们就直接忽略这个新 Bug。

这样一来，Bug 跟踪系统就能让我们始终专注于最重要的 Bug，而不会成为管理负担。

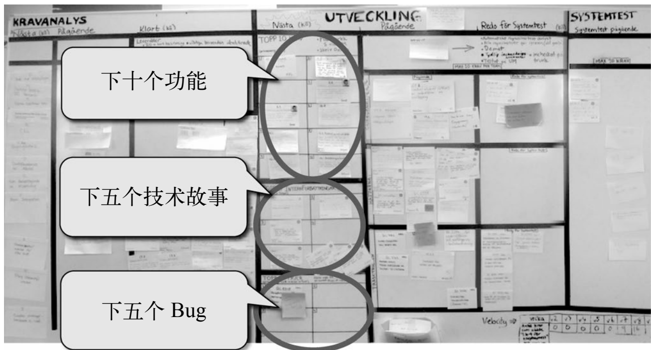
下图是对工作流的总结。

通过控制 Bug 数据库的大小，我们不再需要漫长枯燥的变更委员会会议来管理冗长的、可能永远得不到修复的 Bug 列表。变更委员会会议仍然会召开，不过会议时间短了很多，也更为有效，因为只需要关注边缘案例即可，即只需顾及那些要先进行讨论才能确定优先级的 Bug。



9.4 Bug 可视化

在前 30 个 Bug 中，我们还会确定前 5 个 Bug。这几个 Bug 会写在卡片上，登上项目进度看板。于是，开发团队就有了三个要关注的输入队列：“下十个功能”、“下五个技术故事”和“下五个 Bug”。



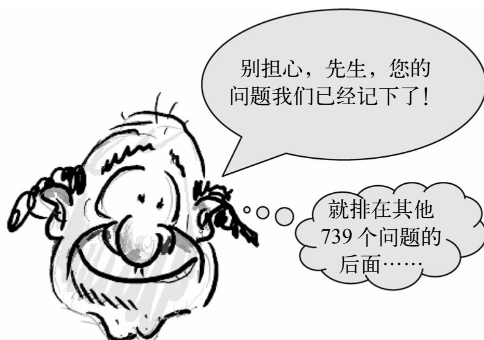
Bug 卡会用红色记号笔书写，这样容易与功能卡和技术故事卡区分。

“下五个 Bug”子栏中 Bug 的重要程度还不足以用粉红色便利贴来表示需要立即修复，但却重要到可以进入“下五个 Bug”列表（一般是在 Bug

跟踪系统前 30 个 Bug 列表中呆过了一段时间)。所以，这种 Bug 很快就会被修复，但不是现在。

等到开发团队有空的时候（通常是刚刚开发完一个功能后)，他们会讨论是从下十个功能、下五个技术故事还是下五个 Bug 中选一个。

限定 Bug 跟踪系统中的数量有利于建立信任。Bug 列表很短，但列表里的 Bug 的确会得到修复，而不是几个月都无人过问。如果有哪个 Bug 一时半会儿不会得到修复（因为其严重程度还不足以进入前 30 名)，那么我们从一开始就要诚实通报，而不是让人苦苦期待。



就要避免上图这样，或者类似情形。

我们仍然要面对的一个需要改进的地方是，我们还没找到一种整洁、一致的方式来将 Bug 可视化。我们还在试验一些方法。测试人员希望清楚了解当前正在修复哪些 Bug，所以他们为此单独设了一块进度板。我们需要综合评估这种做法的优点和额外增设进度板的缺点。Bug 便利贴要往哪里贴呢——是贴在 Bug 进度板上，还是项目进度看板上，还是团队任务板上？我们该多复制几份 Bug 便利贴吗？那些非常小、只需要几分钟就能修复的 Bug 怎么办，我们应该如何避免为此增加这么多工作量？很多问题未有定论，我们还在多方试验……

所以，基本上我们是走过了很长一段路才找到一种解决方案，从而快

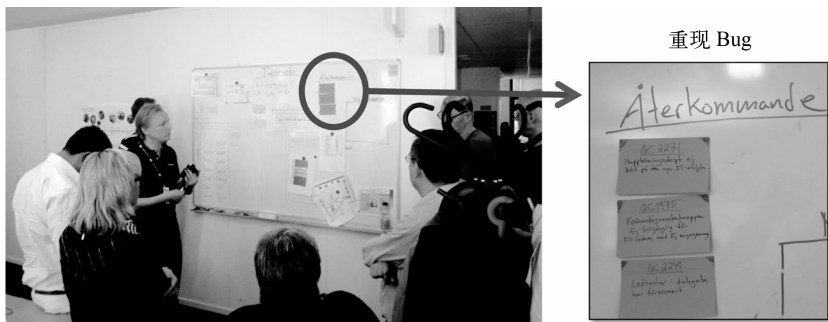
速找到并修复 Bug、改进开发人员和测试人员之间的合作效率、避免在 Bug 跟踪系统中积累太多老 Bug、避免冗长的变更控制委员会会议。不过，我们仍然在尝试解决可视化问题，并不断试验如何才能在进度板上保持适当程度的细节信息。

9.5　预防 Bug 重现

有些类型的 Bug 会反复出现。通常都是些简单的问题，比如用户界面中的标签没有对齐或有错别字。值得思考的是，这些 Bug 当初是怎么进入系统的——是什么样的流程问题在导致这样的技术问题？

为了解决这个问题（而不是只顾抱怨），测试人员在测试任务板上专门辟出重现 Bug （Recurring Bug）区域。还记得我说过 Bug 是写在粉红色便利贴上按障碍来处理的吗？在这里，如果测试人员强烈感觉到某个 Bug 似曾相识，就把该 Bug 贴到测试任务板的这个区域中。这种重现 Bug 的数量限定在十来个。

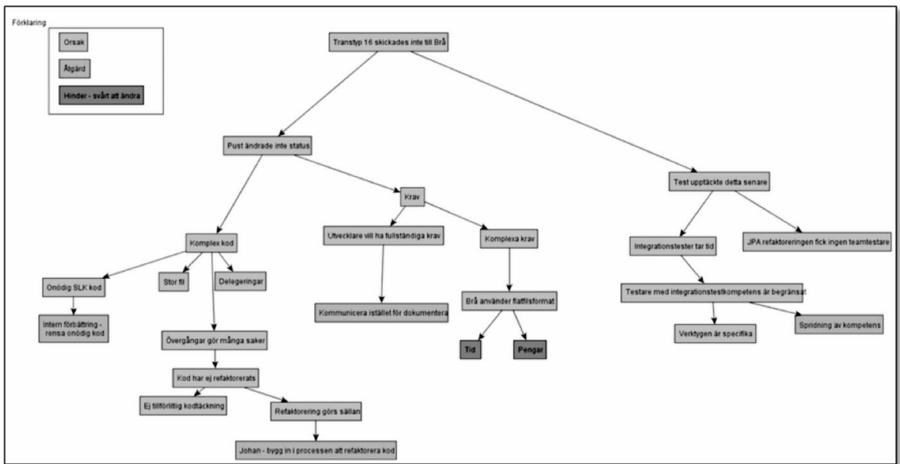
（你现在已经注意到这个原则了吧？所有队列都要限定长度！）



隔一段时间就有一个开发团队会召开缺陷预防会议，他们会选出一个重现的 Bug，分析其根源。这个 Bug 怎么进入代码的？是什么系统问题在导致这种 Bug 反复出现？我们能做什么？Bug 的根源有时与使用的工具有

关，有时与策略和规则有关，有时则与文化问题有关。

如果是比较蹊跷的案例，我们就会画因果图来确定 Bug 的后果和根本原因，并以此为依据，提出切实可行的建议，避免今后出现类似 Bug。比如下面这个例子。



开发团队分析的 Bug 与某些事务有关，这些事务在与另一个系统沟通时丢失了。开发团队提出了如下四项提议，以避免今后出现同类型 Bug。

- 从某个具体模块中移除一些不必要的代码。
- 设置例行程序，确保有更多时间进行重构。
- 多进行面对面沟通，少依赖文档沟通。
- 开发期间与测试人员更加紧密地合作，做得更专业。

用因果图这种方式来做因果分析非常棒，尤其是开始分析类似下面这个例子的诡异问题时。

戴安娜：这个 Bug 怎么出来的？

菲尔：因为我们测试得不对。

戴安娜：怎么会测试得不对？

菲尔：因为我们压力太大。

戴安娜：怎么会压力太大？

菲尔：因为我们已经落后于版本发布计划了。

戴安娜：怎么会落后于版本发布计划？

菲尔：因为上一个版本中的 Bug……

相当诡异，是不是？

杰瑞·温伯格（Jerry Weinberg）表述得非常贴切："之所以有这样的结果，是因为当初就是按这样的方式来做的。"

或者还有一种说法（不知道谁最先说的）："遵循以往的惯例做事，那么不会得到更好的结果。"

无论如何，如果需要了解有关因果图的更多内容，请参见第 20 章。

如果对 Bug 进行因果分析，你就会发现很多 Bug 并不是真正的问题，而是一种症状。产品中的 Bug 是流程有 Bug 的症状。如果你专注于修复流程，产品中出现的新 Bug 就会大幅减少。就好像如果你加强防火，就不需要总忙于四处扑火一样。

所以，下一章将介绍我们是如何连续改进开发流程的。

第10章

持续改进流程

我们的流程绝对不是预先设计好的。我根本不可能事先搞定流程，尤其不可能独自一人完成。即便我能搞定流程，参与项目的人员也不会买账，所以任何事先设计的流程都只会永远停留在纸面上。

与其说我们的流程是设计出来的，不如说是我们发现的。我做的第一件事就是落实流程改进引擎（Process Improvement Engine）。嗯，其实我并没有在工作中用这样的说法，但论实际效果它的确如发动机一般，给我们提供了强大的动力。我们的流程基本模式如下。

- 透明度：在显眼的地方设置看板，向所有人展示项目工作进展。交付物目标明确，人人都能明白。
- 沟通：各个团队内部以及跨团队层面定期召开流程改进研讨会（每周或每两周）。
- 数据：跟踪一些简单的衡量标准，看流程是否有改进。我们衡量的是速率和周期时间（请参见第 12 章）。

我们的战略相当简单：即假设人们天生希望项目成功，天生希望解决问题并改进工作方式。所以，我们需要做的就是创建一个能够促进和鼓励这种行为的环境。

如果人人都清楚我们的目标是什么、我们当前所处的位置，而且有正确的沟通平台，那么大家就会自我组织，朝着正确的方向前进，并持续摸

索出尽快实现目标的方法。

这种激励人们对流程作出革命性改进的思维就是敏捷和精益的基础。

10.1 团队回顾

我们的流程改进研讨会基本上是 Scrum 式的 Sprint 回顾。按照惯例，功能开发团队把流程改进研讨会称作回顾（Retrospective），交叉团队则把这种会议称作流程改进研讨会（Process Improvement Workshop），所以这里我们沿用这些术语。

每个团队每一两周都会召开一次回顾会议，时长三十分钟到两小时不等。有些团队就只是站在他们自己的任务板前开会，有些则专门找会议室开会。有一次我们甚至去了附近一家酒吧开会。会议通常由团队主管主持，不过有时他们也会拉外人来主持（比如我）。

时不时从团队之外找主持人的想法很不错，因为这样会给团队召开回顾会议的方式带来一点变化，并且能让团队主管有机会学习用不同的方式来主持回顾会议。而且这样也能让团队主管专心开会，而不是忙于主持。

找外部主持人一个简单省钱的方式就是让团队 A 的主管去主持团队 B 的回顾会议，或反之。

召开回顾会议的方式可以千差万别，但目标都一样：反省哪里做得好，哪里做得不好，以及需要做出什么样的改变。

改变通常包括以下内容。

- 更频繁地检入代码。
- 改变每日立会的时间或开会的方式。
- 更新代码编写规范。
- 确定一个新的团队内部角色，比如“编译版本主管”（保持编译版本整洁）或“守门员”（保护团队不受干扰）。

团队层面的回顾会议的另外一个重要职能是确定上报要点，即那些不止影响到本团队，还需要与其他团队一起解决的问题和改进建议。这些内容由团队主管记录下来，然后在跨团队的流程改进研讨会上提出来。

10.2 流程改进研讨会

流程改进研讨会基本上是 Scrum 的 Scrum 类型的回顾会议，每个团队和每个专业角色各一位代表（就是每天上午 10 点钟在项目进度板前碰头的那个“交叉团队”)。这是触发影响范围较广的改变的最有效场所，比如，做出影响多个团队的改变和跟进先前改变的结果。

流程改进研讨会的明确目标是澄清并改进我们的工作方式。作为教练，我的一个首要任务就是设立并主持流程改进研讨会，直至这种会议融入到组织文化之中。

最初我们每周四下午 1 点召开研讨会。周四下午 1 点开会纯属巧合——项目人员在这个时段相对能抽出空来。大约一个月之后，我们改为每两周一次，每次还是周四的下午 1 点开始。刚开始的时候会议频率较高，是因为我们经历了成长之痛和困惑，我们需要快速改进不同专业角色之间的协作效率，而这就意味着要做很多尝试。

但每周都开流程改进研讨会则过于密集，我们还没来得及执行上一个研讨会总结出的改变建议，就要准备下一个研讨会。但这样做的积极一面是，频繁的研讨会促使我们快速实施改变，否则下次开会坐下来时说“该死，我们还没来得及实施那项改变”会让人很尴尬。而且，因为研讨会每周都要开，我们必须保持会议简短、专注，这就迫使我们只能优先考虑最重要的改变，并在变革流程中采取较小的步骤。

细想一下，其实研讨会并不是很短，刚开始时 60 分钟，后来不得不延长到 90 分钟，因为我们总是超时。对于每周都要开的会议而言，这个时间相当长。而且我们需要作出的改变相当大，根本不是小步子。回想起来，

我说不清这是好还是坏。我们的确需要作出快速改变（不说别的，让人恐慌的最终期限在一天天逼近）。但这样快速的变革也带来了混乱，尤其是对大多数不在交叉团队的人来说，他们只看到总有改变在发生，却始终没有机会了解相关改变的来龙去脉。

一旦解决了最重要的问题，就可以放慢实施改变的速率，调整到一个适当的程度，所以我们后来就改成两周开一次研讨会。这样感觉更为人性。现在我们可以开九十分钟的会而不再感到紧张（因为不是每周都开），也容易实施一项改变并在下次开会之前看到改变后的结果了。

召开流程改进研讨会时，我会专门把所有的桌子搬开，在办公室中央、项目进度看板跟前把椅子摆成一圈。



这种方式对提高协作水平、增强会议的专注程度有显著效果。大家面对面而坐，中间没有任何障碍物，也没有摆在桌子上的纸张和电脑供人分神。

每一次流程改进研讨会都遵循同样的基本步骤，大体上对应于戴安娜·拉尔森和伊瑟·达比合著的书《敏捷开发回顾：使团队更强大》（*Agile Retrospectives: Making Good Teams Great* [DL06]）中确定的几步。

流程改进研讨会的流程大体如下。

(1) 做好准备：会议开场，设定主题与关注焦点。

(2) 收集数据：回顾上次会议之后发生过什么，包括取得的成果和面临的难点。如果我们有主题，就围绕主题回顾具体数据。

(3) 产生见解：讨论数据及其意义，专注于最重要的难点问题，并确定解决问题的具体方案。

(4) 作出决策：决定要实施哪些改变。

(5) 结束会议：决定谁来做什么，以及下次会议之前需要完成什么。

会议开始时，我会让大家快速轮流发言，好让每个人都开口，比如，"用一个词形容你现在的感受"或者"你希望这次会议的最大收获是什么"。

然后，我会提醒大家本次会议的目的，并提到今天会议的关注焦点是什么。有时我们会有具体的关注焦点（如动机、度量、协作、测试自动化等）；有时除了改进我们工作方式这个大目标之外，并没有具体的关注焦点。

接下来，我们总结上次会议以来发生的主要事件，并讨论上次会议有关决策的完成情况。

然后我们快速总结过去几周哪里做得好，有哪些较为积极的改进。有时我让参会人员把发言内容写在便利贴上，然后贴到看板上；有时他们说，我往白板上写。总结并肯定过往的改进对激励进一步的改进非常重要。

然后我们快速总结目前的难点和挑战。如果有很多问题（通常都会有很多），我们就会排出优先级，通常是用计点投票表决或类似的方式。计点投票表决是给会议室内每个人三个小圆点，大家可视自己眼中的重要程度选出三项贴上圆点。

下面这个例子显示有两个栏的便利贴："成果"与"挑战"。

延伸讨论

任何一位较为正统的精益开发热心人士这时可能都会指责我们："这些都太主观，太感情化！流程改进必须通过量化、客观的数据和报告来推动！"

我个人对此持有不同意见。这种复杂的产品开发是一个由有创造力的人来完成的创新过程，而且最重要的要素就是动机。在这种环境下，直觉比硬性的度量数据更有说服力。如果我们感觉是个重要问题，则无论是否有度量数据去证明，它都极有可能是个重要问题。而且直觉的优点是，感觉到有问题通常是问题可能会发生的先兆，而硬性的度量数据只有在问题发生以后才能显示出了问题。

的确，我们也采用了硬性的度量数据（参见第 12 章），而且有时这些度量数据会让大家意识到出现了问题（也就是著名的"哦，糟糕！"时刻）。不过我们主要用度量来支持流程改进，而不是推动流程改进。

好了，我们说到哪里了？哦，对，所以我们列出所有难点并予以排序，然后选出一两个最重要的难点，作为本次会议的关注焦点。然后我们分成两三组，分别讨论并分析问题，寻找可能的解决方案。

有时解决方案会相当简单明了。对于更为复杂或重复发生的问题，我们用因果图或类似的工具（第 20 章介绍因果图）分析根本原因，然后提议做一些专题研究，以期能为下次流程改进研讨会带来有用的度量数据，或提议由相关人员另行研讨、解决问题。

通常情况下，分组讨论都能产生几项具体的提议或备选方案，我会把这些内容都列在白板上。默认的方案始终是保持现状（“不要做任何改变”），并提示大家如果本次会议不能就新方案达成一致意见，会出现什么样的后果。

对于每一项备选方案（包括保持现状的方案），我们都会进行头脑风暴，列出最明显的优势和劣势。多数情况下，这种快速分析都能揭示出哪个方案最好，于是我们会形成共识，实施该项方案。对优劣难分的其他选项，我们会通过快速大拇指投票来看看大家对每个方案的感觉如何。



“我赞成这个方案。”



“这个方案不够好，不过还能接受。”“我没什么意见，听大家的。”“我现在决定不了。”



“这个方案不好，我不支持。”

有时我们会采用“五指表决”的方式。“五指表决”类似于大拇指投票，不过更加精确。每个人不是只用大拇指，而是通过伸出几个指头来表示不同的含义。


“这个方案棒极了！”


“这个方案很不错。”


“这个方案不够好，不过还能接受。”“我没什么意见，听大家的。”“我现在决定不了。”


“这个方案不好，我不支持。不过也许谁能说服我。”


“绝不支持！”

这两种方法的要点是，大拇指倾斜或伸出三根手指是意见是否一致的分界线。任何方案，只要参加研讨会的每个人都支持（或赞成），就值得实施。与会人员不一定非要喜欢该方案，但每个人都会接受并支持该方案。这就是意见一致的含义。

这种类型的一致表决通常都能够清楚揭示出最佳方案。如果备选方案很多，且结果不够明了，我们就会先把那些有人不能接受的方案划掉——也就是说，所得投票中有只伸出一两根手指或大拇指朝下的方案。那些方案没有获得一致赞成，就意味着基本上被否决了。然后，我们会审视剩下的方案，把获得最高支持率的方案挑出来。如果我们无法就任何方案达成一致意见，默认的方案就是保持现状。

偶尔也有两个方案拥有同样的支持率，这时我们或者就任选一个，或者选容易做的，或者来一次一决胜负的投票（“假定要从方案 D 和方案 E

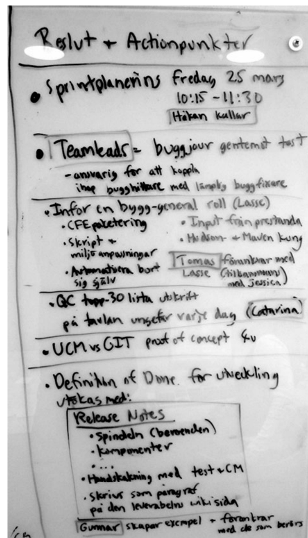
中选择，你会选哪一个？”)。作为会议主持人，我会确定每一种情况下的决策流程，来避免大家又为了决策流程而讨论不休（一群聪明人在一起很容易如此）。与会人员如果不喜欢主持人选定的决策流程，也可以提出反对意见。

这些达成一致意见的方法听上去很低效，但其实多数情况下都很快、很有效率。个别时候效率不高，则意味着有必要再做深入分析工作。

作出有关流程改进的决策意味着作出改变。既然我们是在跟人打交道，改变就意味着可能遭遇阻力，尤其是来自那些没有参加会议的人员。我们争取每项改变都达成百分百一致意见（即与会的所有人都意见一致），从而大幅降低了阻力风险，极大地保证了改变能够顺利进行。因此，在达成一致意见方面多花几分钟就能够获得巨大回报。

我们严格控制研讨会时间，通常都在 90 分钟（包括极其重要的五分钟休息时间）。研讨会最后留出十分钟左右的时间，总结已作出的决定（在白板上列出），并确定具体的行动——谁什么时候要做什么。

下面是一个例子。



在此次会议上我们做出了很多决定（比平时都多）。前两项如下。

- 尝试实施“迭代规划会议”概念，让不同专业角色的人员一起协作，调整并细分用户故事，然后决定哪几个可以拉入项目进度板上的“下十个功能”栏。更多内容请参见第 13 章。如果实施效果很好，我们就继续开迭代规划会议，可能两周一次（后来看，实际效果的确很好）。
- 每个团队明确指定一位“Bug 联系人”，测试人员发现 Bug 后，不知道谁具体负责修复的时候就来找这位联系人。如果未指定 Bug 联系人，则默认由团队主管担任。

这两项改变实施起来较为容易，因为其目标是解决大家目前遇到的现实问题，而不是针对理论问题的理论性解决方案。

记住，会议宣称的目标是澄清并改进我们的工作方式。有时我们不用作出任何改变，只是澄清我们的现行流程——也就是说，消除混淆与误会，给出明确的描述和说明，让与会人员可以向他们各自的团队传达。有一次我们就专门澄清了验收测试、系统测试和功能测试的具体含义。

10.3 掌控改变速率

每周的流程改进研讨会都会带来一波改变，而且多数改变的结果都很积极。不过，过了一段时间后，我们意识到改变太多太快了。

这是一个很有意思的问题。在我服务过的多数组织中，问题都是几乎没有什么流程变化——人人都固守现行的低效流程。这里却是相反的问题，我们作出的改变太多，而通常一项较大的流程改变需要数天，有时甚至数周才能在一个 60 人的项目中完全见效。因此，当前一波改变的尘埃还未落定时，交叉团队又引入新一波的改变，导致很多团队成员很迷惑（甚至很沮丧）。

于是，我们引入了一点官僚做法来降低改变的速率。如果有人需要作出不止影响本团队的改变，就需要提交一份流程改进提案。这是精益 A3 问题解决流程的轻量版[①]。

流程改进提案模板会迫使你思考为什么要作出一项改变。

- “你要尝试解决什么问题？”
- “这项改变会影响到谁？”
- “执行这项改变涉及什么步骤？”

思考这几个问题非常有利于确定这项改变带来的价值与成本。图 1 是一个流程改进提案的真实例子。

建议书：对客户更有价值的用户故事

为什么？我们要尝试解决什么问题？

- 难以从客户角度获得项目进度板概况，很多用户故事太小，无法交付

这项改变会影响到谁？

- 需求、开发和测试团队

实施这项改变的步骤是什么？

- 更新“可供开发”的完成定义，增加“用户故事对客户有价值”这一句。
- 审视进度板，确定哪些用户故事太小而没有价值。将这些用户故事组合成大一些的用户故事（只要不超过中）。

例如，

没收
大

没收
登记没收
中

没收
删除没收
小

说明/常见问题

进入开发阶段的用户故事必须满足以下要求：

1.规模为小或中

2.只要不违反规模规则，应当尽可能对客户有价值。

需求团队确保“可供开发”栏中的每张卡都是一个对客户有价值的用户故事（无论规模大小）。不过，进入开发阶段前的规模必须为小或中。

问题：如果用户故事规模为大，而且必须作为整体交付才对客户有价值，这时应当怎么做?

- 将其分割成较小的用户故事（新卡片），规模为中，但每张卡片的客户价值都需要尽可能高。
- 在每张卡的顶部用蓝色粗笔写下所属的较大功能的标题，便于卡片归类。

图 1　流程改进提案示例

① 参见 www.crisp.se/lean/a3-template。

此项提案的主旨是保留对客户更有价值的功能，还建议完全不应将那些估计规模很大的功能拉入开发阶段，因为这类功能很有可能会膨胀并堵塞流程。反之，应当将这类大功能分割成较小的可交付单位。然后，等分割完后，如果这些较小的功能单独而言对客户没有价值，那就需要用蓝色粗笔在功能卡顶部写下标题，说明几个小功能组合起来才是一个完整功能。这样做有助于从版本发布角度把几个小功能组合在一起。

任何人都可以提交提案。通常写提案的人会出席流程改进研讨会，陈述提案并回答问题。我们的模板基本上是把一项提案转变为以具体改变为目标的小型商业案例，这样更容易对其进行优先排序并作出决策。

引入这个小模板的目标是让我们控制改变的数量。所以，如果收到四项提案，我们可能只会实施其中的一两项，即便四项提案都非常棒。要拒绝采纳一项非常棒的流程改进提案非常困难，但我们也意识到必须控制要同时实施的流程改进举措的数量。否则，我们就会造成更多的混乱，反而会抵消流程改进所带来的益处。

我们甚至考虑过单独设一块流程改进进度看板来控制变更提案的数量，显示目前正在实施哪些改变。这对跟进实施进度也会很有用。但这样一来，我们就需要找地方安放这块进度板，然后还要随时更新。嗯……还是算了。

如你所见，看板并没有很多具体规则。很多具体决定，比如一个项目使用几块看板等，完全取决于你自己。这也是看板的迷人（和痛苦）之处——非常灵活，你需要随着项目的进展摸索出最合适的方式。不过，你只需要遵循这条经验法则：“犹豫不决之时，就选择最简单的方案。”

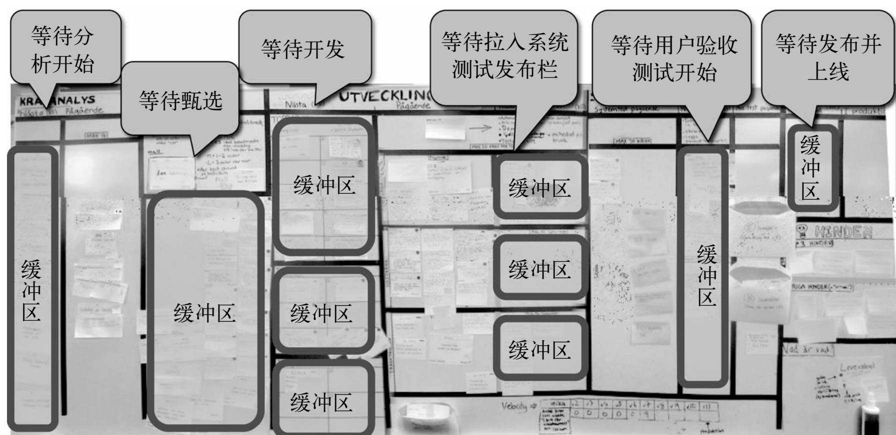
第11章

管理在制品

在任何一种工作流中，将工作状态与等待状态区别开来都很有用。当载有你信件的邮车正在运输途中时，装有你信件的信封就正处于工作状态（因为邮车正在运送它）；等信件躺在你家邮筒中时，信封就处于等待状态了（因为什么事都没有发生）。

之所以需要邮筒，是因为我们不会一直站在那里等候邮车抵达以收取信件。邮筒就是缓冲区——供待处理事项停留，等待下一个流程步骤。

现在，如果你仔细研究项目看板，就会看到只有四栏表示在制品（WIP）。其他六栏都是缓冲区（或队列），如下图所示。



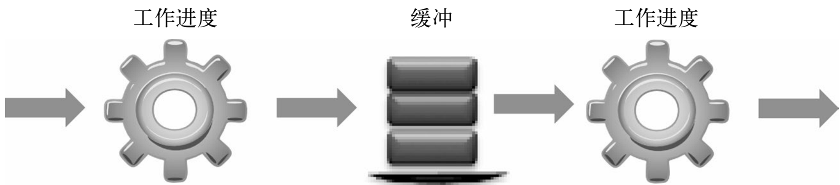
我们在栏目标题旁写下“队列”来突出显示缓冲区。

将缓冲栏与 WIP 栏区别开来非常有用，因为呆在缓冲区里显然都是浪费——那些功能只是停留那里，对吧？缓冲区越大，等待处理的时间就越多！

再想想邮筒。邮筒里堆积的信件越多，要读到某封特定信件需要的时间就更长。这就等同于浪费。你会因此而需要更大的邮筒，而朋友们会因为始终得不到你的回复而不开心，寄给你的账单也可能得不到及时支付。

然而，有时我们的确需要一个小缓冲区来保证两个流程之间的顺利衔接。例如，功能的开发速率与下一步的系统测试速率并非完全合拍，这时的缓冲则可视为“必要的浪费”。

设想一下，有一家工厂的工件（比如信封）如下图所示从左边流向右边。



除非两台机器能够以同样的速率严丝合缝地前后衔接，否则就需要一个缓冲地带来消化两边的变动。

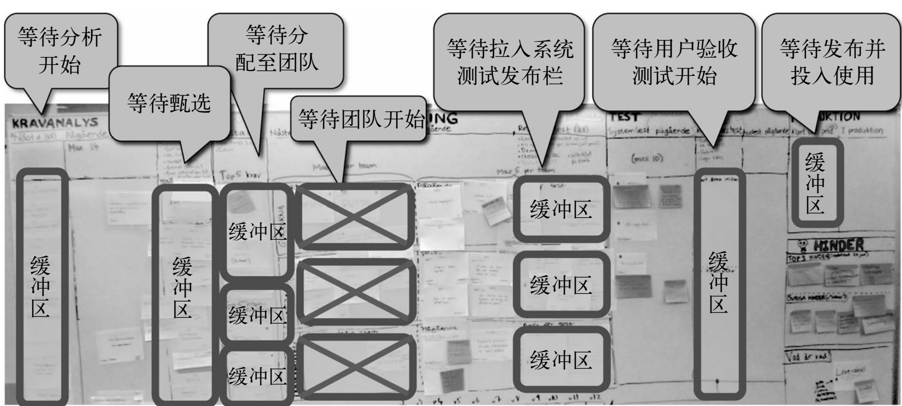
不过，随着流程的改进，对这种缓冲区的需求就会减少。在进度看板上将缓冲区清晰标示出来，我们就更有可能不断自问：是否真的需要所有这些缓冲区？我们能做什么来减少缓冲区的数量？

下面这个例子是一个老版本的进度看板，我们在需求和开发之间还有一个名为“等待团队开始”的缓冲栏，即图中用叉标出的地方。

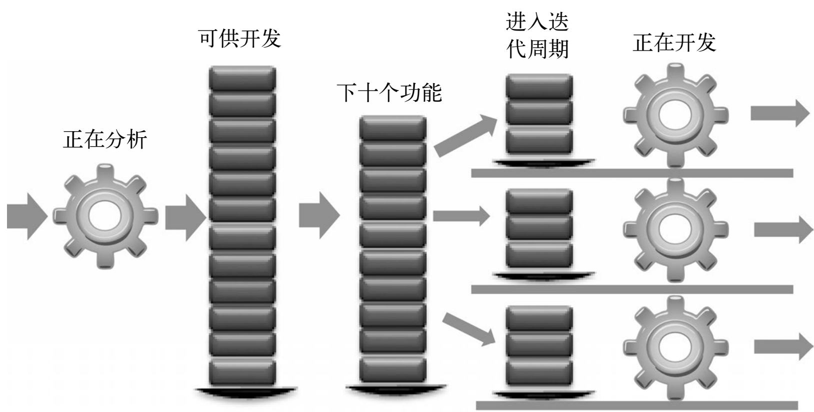
当时我们设这样的缓冲区是因为各个团队采用了更像 Scrum 的 Sprint 开发模式（表示迭代的 Scrum 术语)。针对每一个新的 Sprint，各个团队都会召开 Sprint 规划会议，并确定一组具体功能。相应的功能卡会被拉入“等

待团队开始”栏。于是，在需求和开发之间，我们就有三个缓冲区：

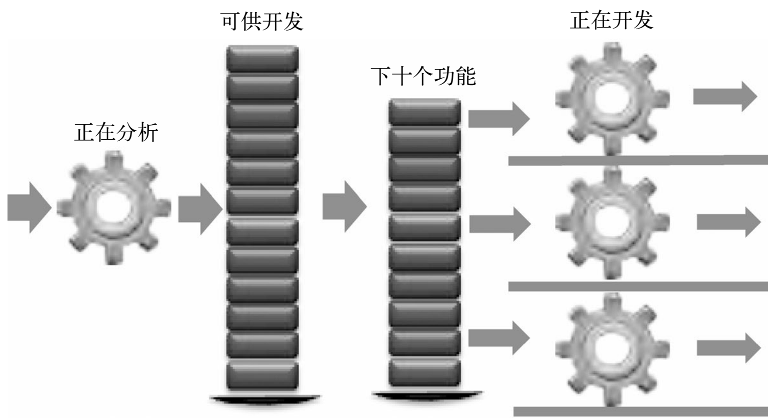
- 已通过分析得到确定但还未选入“下十个功能”栏的功能
- 已拉入“下十个功能”栏但还未被具体开发团队接手的功能
- 已经在具体团队手中但开发工作还未真正开始的功能



如果用工厂的生产流程来表示，这些缓冲区将如下图所示。



当时我们注意到一点，我们浪费了太多时间争论哪个功能应当进入哪个缓冲区。于是干脆把第三个队列去掉，让开发团队直接从前十个功能中挑选，而不是让一批批功能进入 Sprint 等待开发。这样就减少了混乱，改善了工序流动。



这就提出了一个专业化分工问题。如果某个团队负责开发具体功能领域，例如，与 X 系统集成，那么让这个团队来负责实施需要与 X 系统集成的所有功能就最有效率，因为他们已经熟知系统 X 是怎样工作的。

不过，这并不意味着该团队需要将所有与 X 系统相关的功能都揽下来。虽然该团队是 X 系统相关功能的最佳人选，但如果这个团队已成为流程的瓶颈，我们就应该考虑让其他团队来出手救援的可能性。

所以，尽管各个开发团队都直接从下十个功能栏中拉入功能卡，他们却在以一种很聪明的方式进行：开发团队在开发同步立会上互相交流，讨论如何最佳利用各个团队的现有能力。

为协助这一流程，我们在项目进度看板上用团队的头像磁贴来表示。团队可以用他们自己的头像磁贴来标记“下十个功能”栏（甚至流程中更早阶段）中的某张功能卡，表示“我们团队开发这项功能最合适”。这样，大家就知道涉及该项功能的时候该去找谁，而且其他团队在挑选下一个开发目标时也会三思是否应将该功能卡拉入。



11.1 采用在制品限额

进度看板每一栏的顶部都写有一个数字，这是在制品限额（WIP limit），如“每个团队最多五个功能”。



在制品限额旨在避免同时做太多工作，避免让下游流程负载过重。如果测试人员手头已经有太多工作，我们就不希望开发人员再持续不断地开

发新功能，给测试人员增加负担，而是希望他们腾出手来帮助测试人员尽快完成测试。在制品限额如同一个警告信号，不等问题失控就引起大家的重视。

这就好像是打印机。一般的打印机在制品限额是一次一张纸，如果卡纸，你就需要打印机立即停止打印并发出警告信号，对不对？无论新的打印任务有多紧急，已经卡纸的时候你都不会再往里添乱，否则只会让问题更严重。

我们几乎在进度看板上的每一栏或每几栏的顶部都用红笔写下在制品限额。“每个团队最多五个功能”的在制品限额是指，如果一个团队已经在开发五个功能了，那么只有当其中一个已完成并开始系统测试时，他们才可以再接手下一个新的功能。

这种做法我们画了一些漫画，请参见 17.5 节。

11.2　为什么在制品限额只适用于功能卡

我们的在制品限额只适用于功能卡。技术故事和 Bug 修复不包括在在制品限额内。

不包括 Bug，是因为 Bug 通常都相当紧急，而且都相当小。此外，我们还未摸索出在进度看板上以统一方式处理 Bug 的好方法。有时 Bug 会在进度看板上标出，有时却没有，我们现在还不希望就此制定太多的规则。

不包括技术故事是因为以下原因。

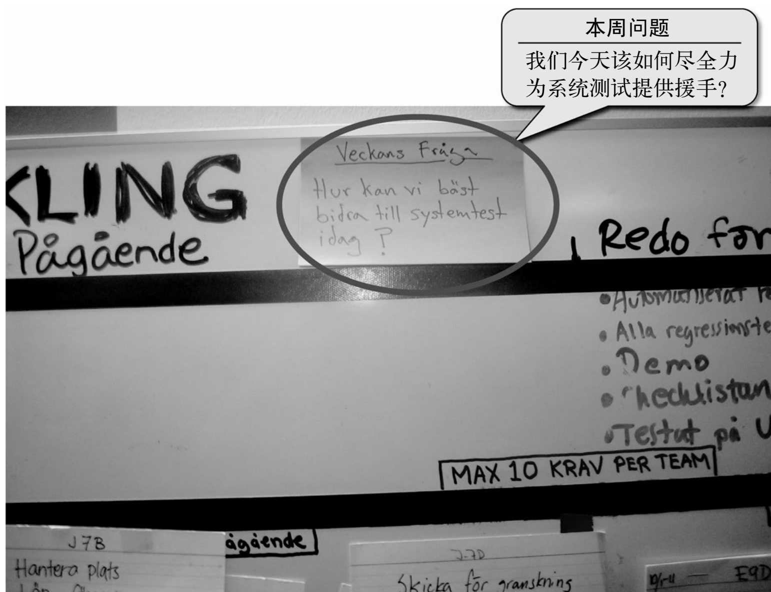
设置在制品限额的一个原因是避免让下游流程负载过重。开发新功能就肯定意味着增加测试工作量，所以如果我们开发功能速度太快，就会造成测试负载过重。

而技术故事却恰恰相反，技术故事有助于消除下游瓶颈。我们很多技术故事都与测试自动化和基础设施改进有关，而这两项工作都能提高质量，让测试人员的工作更轻松。

当系统测试成为瓶颈时，测试团队会专心完成当前一轮的系统测试工作，这意味着他们还需要几天的时间才能将下一批功能卡从“可供系统测试”栏拉入“正在进行系统测试”栏。而每个团队五个功能的在制品限额同时适用于“正在开发”和“可供系统测试”栏。这样做的结果是，当系统测试成为瓶颈，开发团队的在制品就会达到限额，因为他们已经开发好的功能还要继续卡在自己手中，直到系统测试团队把功能卡移走。

那么，在制品限额已满的时候，开发人员该做什么？

他们应当做任何能够协助或减轻系统测试的工作！实际上，有一次系统测试成为严重的瓶颈时，一位测试人员将其作为“本周问题”贴在进度看板上，提醒大家每天在项目同步立会上都看看这个问题。这一做法非常有效！



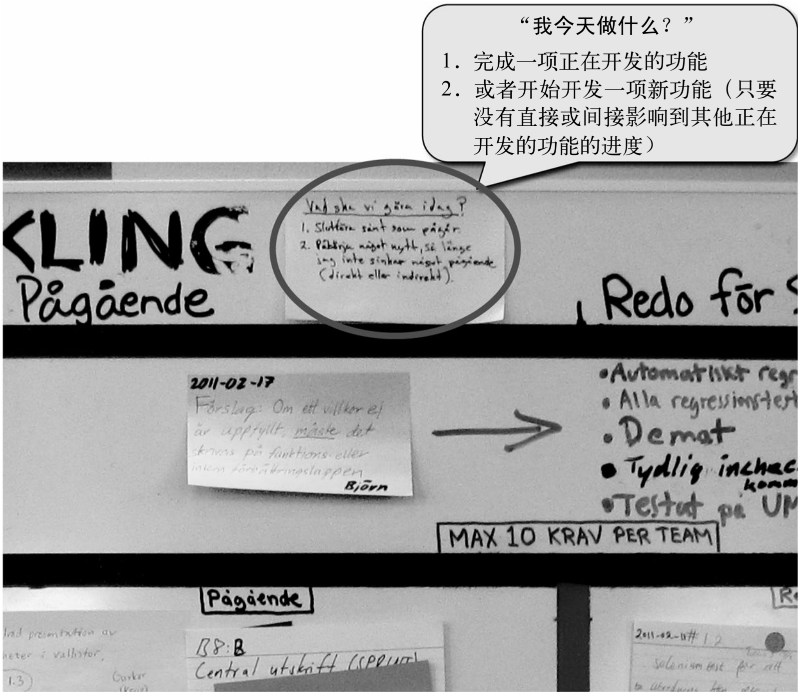
为测试提供援手的一种方式是调动人力去做手动测试和 Bug 修复，还有一种方式是开发自动化程度更高的测试用例并改进测试基础设施。这些工作都用技术故事来表示。这就是为什么我们未将技术故事纳入在制品限

额范畴的原因，我们希望鼓励团队成员在在制品限额已满时花点精力来处理技术故事。

有一段时间几乎所有团队都完全专注于测试自动化工作，项目进度看板上也只有带绿色圆点的卡。这个例子极好地说明，将看板与在制品限额结合运用是如何促进自我组织并消除瓶颈的。

在制品限额只适用于功能的另一个原因是，这样做与我们衡量工作的方式是一致的。在衡量周期时间、计算速率时，我们只包括了功能卡（更多内容请参见第 12 章）。基本而言，技术故事和 Bug 目前都不在在制品限额和度量的控制范围。当然，以后会改变的……

至于“本周问题”，实践证明这个概念非常有用。我们后来扩大了问题的定义，让其更具普遍性（瓶颈不一定是系统测试环节），并将其作为排序依据，来决定“我今天做什么”。



采用在制品限额的一个后果是，有时大家会没事做。更准确一点说，在制品限额有时会强制大家去做与常规工作内容不同的事（例如，帮忙进行测试，而不是开发新功能）。在这种情况下，项目进度看板上的“我今天做什么”列表就很有用了。

实际上，这句话捕捉到了在制品限额的精髓：努力完成一件事，而不是努力开始做另一件事！

那么，我们在完成一件事上做得怎么样呢？这就是下一章的内容了。

第12章

捕捉并使用流程度量

流程度量非常有用，可以找出哪里需要改进，看出我们所作的变更是否带来了积极的效果。对于从宏观上规划版本发布大纲也很有用。

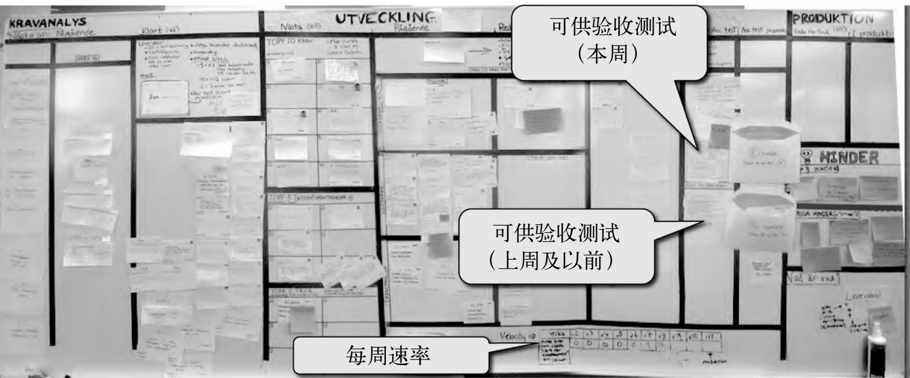
我们追踪以下两个流程度量：

- 速率（每周功能数）
- 周期时间（每个功能的开发时间）

我们完全手动捕捉这两个度量。这一做法之易，让我很诧异为什么不是所有项目都采用。

12.1 速率（每周功能数）

对于速率（生产能力），我们只在每周结束的时候计算有多少张功能卡可以进入“可供验收测试（本周）”栏。我们在进度看板底部的速率栏内记下这一数字，然后将功能卡移入“可供验收测试（上周及以前）”栏，表示这些卡已经计过数。

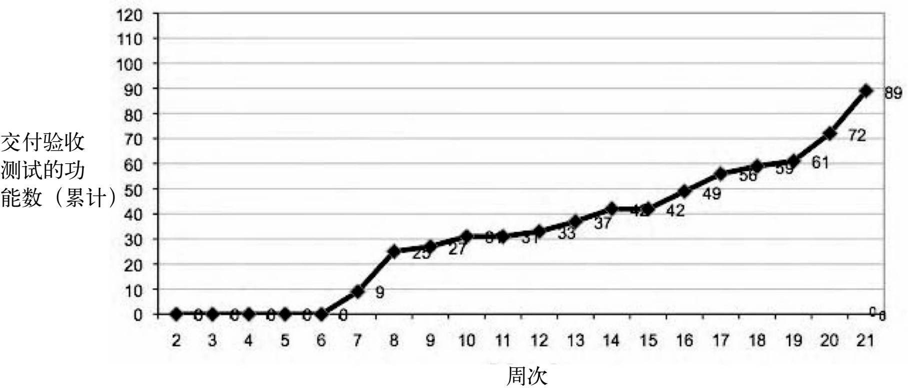


下图是进度看板速率记录栏的特写。

Vecka	v10	v11	v12	v13	v14	v15	v16	v17	v18
Antal nya funktioner som nått till 'Redo för AccTest'	4	0	2	4	5	0			

↑ Prodsättn.

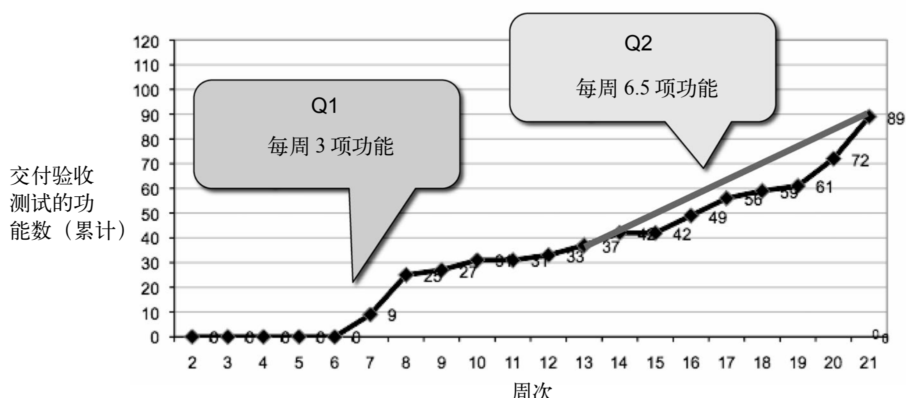
我们用这个数据生成一个简单的燃尽图，显示每周所完成功能的累计数量。



这一数据在很多地方都很有用。首先，可将其用作现实检查工具，检验版本发布计划是否切实可行；其次，根据这一数据来预测截至某个时间点大致可完成的功能数（参见第 6 章）。

燃尽图也可用来凸显问题。例如，速率在开始衡量的头几周为 0。各个团队都在努力工作，但系统测试因为各种原因成为瓶颈，于是所有的功能都在排队等着测试团队测试。人人都可以看到进度看板上卡片堆积，人人都可以看到一周一周过去，速率仍然为 0。这就会给大家一种紧迫感，让开发人员逐渐转移工作重点，向测试团队伸出援手，而不再只是专注开发新功能，继续给测试团队雪上加霜。

最后，燃尽图还用来直观显示流程的改进。例如，在下图中我们可以看到，第二季度（Q2）比第一季度（Q1）的平均速率（曲线的斜度）翻倍了。这种可视化结果有助于激励所有人都去不断改进流程。



不过需要注意的是，这种类型的统计数据要谨慎使用。在创建此图之后刚开始的几周时间里，曲线平缓，这是因为团队在忙着内部改进工作，所以速率为 0。因此，更为切实的估算结果是，速率大概提升了 50%，而不是翻倍。

我们也考虑过衡量技术故事的速率，这样就能将客户需求和技术故事这两方面的工作时间分配以可视化方式显示出来。两种速率结合起来就可以反映出我们全部的工作能力，这样燃尽图曲线会更平滑，规划工作也会更轻松。

12.2 为何不使用故事点

这时候你可能在想，我们怎么会只计算功能数量，那功能大小呢？衡量速率的时候不应该考虑功能的大小吗？如果与 Q1 相比，Q2 的速率翻倍了，是不是真的就意味着我们的工作效率提高了？Q2 完成的功能数量多是不是因为那些功能都小一些？

这种过分简单化的速率有没有误导性？

理论上来说，有。不过，从实践上来说，功能大小的分布其实相当均匀。我做过一个小实验， 根据预估的大小给每个功能分配一个权重，所以小=1 公斤，中=2 公斤，大=3 公斤。多数敏捷团队会将其称为故事点，即开发一个功能所需时间的相对预估值。

这里打个比方。假设我在垒砖块，再假设我希望测算自己垒砖快的速率。我的速率大概是每分钟可以垒 10 到 15 块砖，这个数字不够精确。不过别急，砖块的重量也不相同啊！如果我们衡量每分钟垒的重量，而不是每分钟的砖块数会怎么样？

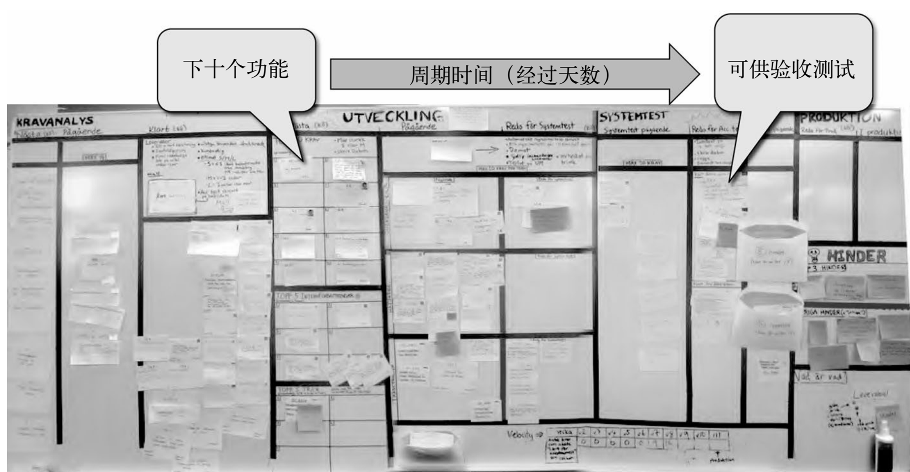
这样或许能得到一个更平滑的速率，因而就更容易预测我今天能够垒多少砖块。

那么，我们要称每块砖的重量，然后计算我每分钟能垒多少公斤的砖块，也就是大概每分钟 20 ~ 30 公斤。等一等，这个数字比之前每分钟 10 ~ 15 块砖的数字没精确多少啊。两种情况下都是 X ± 50%！只要砖块大小大致均匀，再为了计算速率去称每块砖的重量就没有多大意义。只需计算砖块数量就可以了。

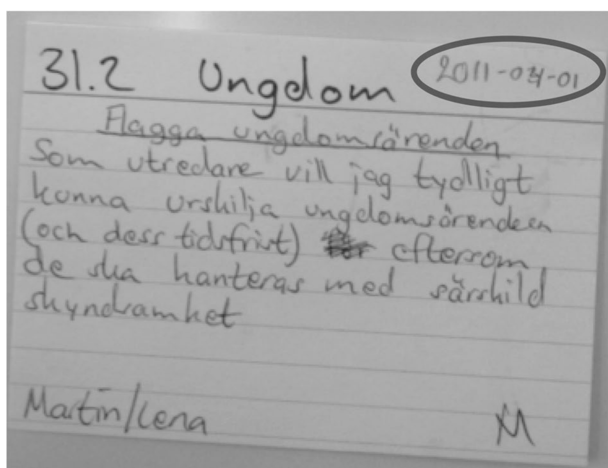
这跟我们的项目情况完全一样，我不过是制作了用公斤数而不是功能数量来表示单位的燃尽图，得出的曲线形状与之前无异。提高精确度并未增加多少价值，所以估算故事点就是在浪费时间。

12.3 周期时间（每个功能所需时间）

我们衡量的另外一个标准是周期时间（Cycle Time，或生产周期）。周期时间是指完成一项工作所需的时间，或在我们项目中更具体的是指，X功能卡从“下十个功能”栏移到“可供验收测试”栏需要多长时间。



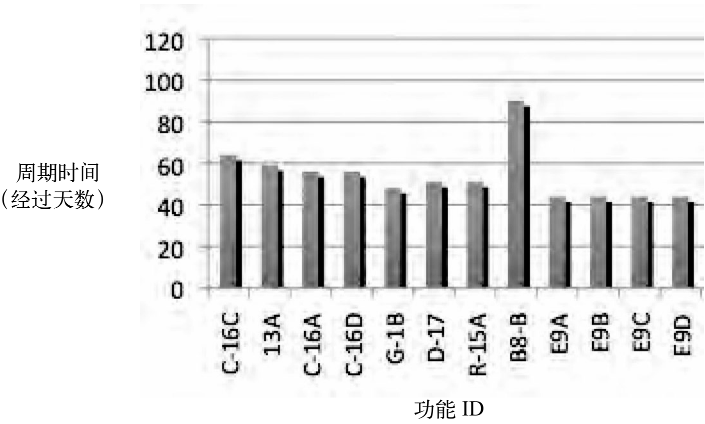
周期时间也很容易衡量。每次选好一张功能卡放入“下十个功能”栏，我们就在卡上记录下开始日期。



每次一张功能卡到了“可供验收测试”栏，我们就记下完成日期，然后在电子表格上记录下开发该功能所经过的总天数。

Leverabel	start	slut	Genomströmningstid (dagar)
6A	2011-02-01	2011-02-17	16
6A	2011-02-01	2011-02-17	16
26B	2011-02-03	2011-02-24	21
26A	2011-02-03	2011-02-24	21
B2.2	2011-02-08	2011-02-21	13
25.2.1	2011-02-08	2011-02-18	10
25.2.2	2011-02-08	2011-03-25	45
B8-A	2011-02-10	2011-03-09	27
T23.2	2011-02-10	2011-02-23	13
16J	2011-02-20	2011-03-28	36
A-4A	2011-03-08	2011-03-30	22

然后，我们用控制图来显示结果，每个柱子的高度都表示完成一个具体功能所需的时间。



这张图可用来预测完成一项功能所需的时间。用这张图来制造一种敬畏感也非常不错，因为多数人都不清楚完成一项工作到底要需要多久！每当公司将这种数据可视化，几乎都会达到这种让人震惊的效果。



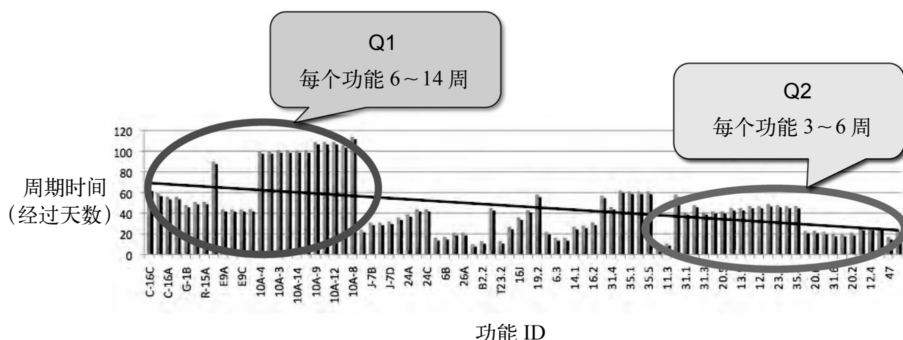
小乔爱问：
你们为什么不全程衡量周期时间，直到产品上线？

因为权利不在我们手中，我们须要大致每两个月发布一次。发布日期是固定的，每次发布之前的最后一两周都是验收测试阶段，真正的用户会过来试用系统。因此，只要一项功能到了“可供验收测试”阶段，就一直处于该阶段，直到发布周期结束。这是瀑布式开发流程的遗留问题……

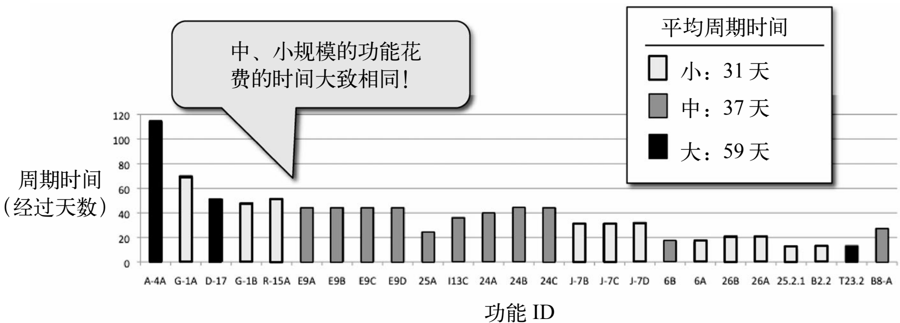
我们计算周期时间只截至“可供验收测试”阶段，因为这部分流程处于我们的控制范围内（因而可以优化）。而且，根据以往的经验，验收测试阶段很少会发现严重问题。所以，通过衡量截至“可供验收测试”阶段的周期时间，我们其实已经覆盖了工作流中最具风险的部分，这对我们就已经足够了。

相当典型的是，经过的天数（周期时间）通常是实际工作时间的 5 到 10 倍。所以，如果一项功能实际开发时间是 3 天，那么按日历计算，总共就需要 20 天左右。造成这种差异的原因通常包括：多项任务同时进行、缓冲冗余以及功能卡排队等候进入流程下一步等。

好在一旦你了解了周期时间，再运用在制品限额等技术将其缩短就不是难事了（请参见第 11 章）。下图的趋势线显示，我们的周期时间在几个月内就缩短了一半。



我们还注意到一个很有意思的现象，即功能的大小与周期时间没有相关性。下面这幅图中，各项功能的大小都按最初的估计以不同颜色标记。



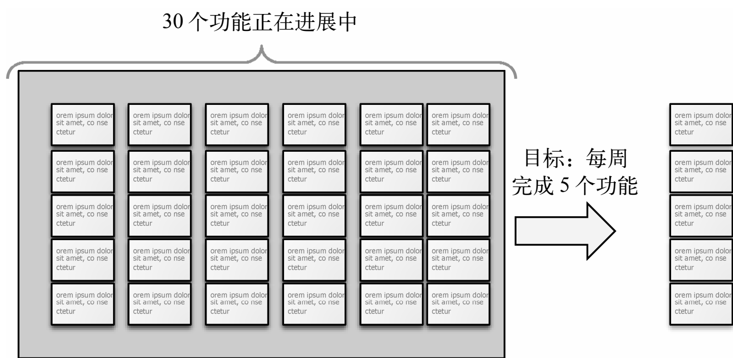
如图所示，有些小功能用了大概 7 周的时间，而有些大的功能也才用了两周时间。事实表明，功能大小并不是影响周期时间的主要因素，其他因素，如专注程度以及是否有骨干人员可用，则更为重要。

有一次，我们仔细审视了这个数据，并设定了几项有挑战性但切合实际的目标，以便引导改进工作。

- 稳定速率：每周的速率应大致相同，而不是分布不均。这样我们的瓶颈应当少一些，版本发布规划就可以更轻松，总体上的工作流程也更顺畅。
- 提高速率：我们目前的平均速率是 3，目标为 5。
- 缩短周期时间：我们的平均周期时间是 6 周（不过缩短得很快），目标为 2 周。

在制定这些目标的时候，我们意识到一个有趣的问题。假设我们实现了前两个目标，达到了每周 5 个功能的稳定速率，这对第三个目标缩短周期时间会有什么意义？

我们的数据（和照片）显示，通常情况下，每天的项目进度看板上都有大约 30 个功能处在不同进展状态。

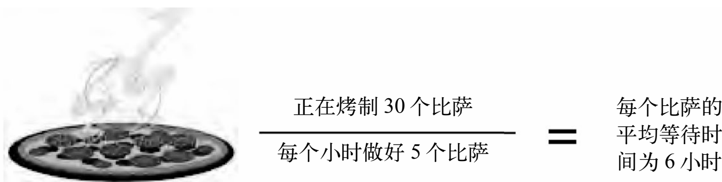


这就意味着平均周期时间是六周！

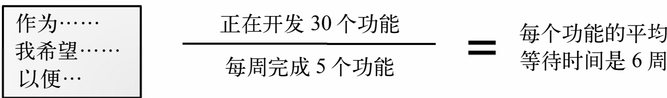
假设你在一家比萨店用餐，他们每个小时能做好 5 个比萨。你得等多长时间能吃到比萨？

12 分钟？

如果比萨店里还有其他三十位顾客都在等着吃比萨，那就不可能。在这种情况下，平均等待时间就是 6 个小时！



这个计算方法同样适用于我们的功能开发。

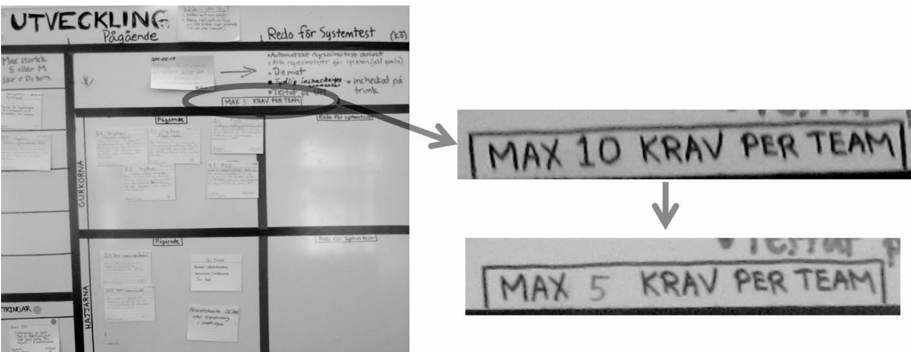


这就是排队理论中的利特尔法则[①]。这是无法避免的。

① http://en.wikipedia.org/wiki/Little’s_law

那么，我们如何将周期时间从 6 周缩短到 2 周？答案是，要么将速率提高 3 倍（增加时间和人力），要么将在制品数量除以 3。你认为哪个方案的成本比较低廉？完全正确！

所以，开发团队将他们的在制品限额从 10 个功能减少到 5 个功能。



虽然这样并没有减少到三分之一，但这个缩减数量仍然相当可观。减少在制品数量时，需要考虑到的一点是，如果减得过多，就可能出现其他副作用，比如大量人员空闲下来。这又会对速率产生负面影响，从而又增加了周期时间。所以，我们得找到一个平衡点。

缩减的目标是设定一个较低的在制品限额，保证所有人员相互协作，问题能够得到暴露——但不至于低到一下子暴露出所有问题（这只会让人产生沮丧情绪，造成流程不稳定）。

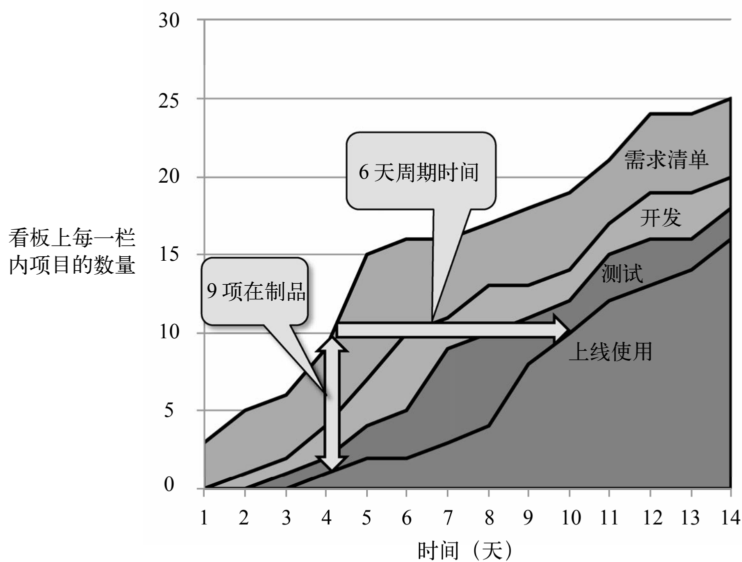
我们还没有达到每个功能两周时间的目标，但这并没那么紧要。设定目标是为了让我们朝着正确的方向前进。我们已经将周期时间缩短了一半，而减少在制品数量就是帮助我们实现这一成就的诸多工作之一。

有了度量，就可以表明我们的前进方向正确无误，这一点很重要。

12.4　累计流量

看板的一大优势在于可以实时显示瓶颈位置，不过显示不了历史趋势。

累计流量图是看板圈内用来形象化展示历史瓶颈的常用工具。每天数一数每一栏中都有多少项，然后用下图这样的累计流量图显示出来。



图中的不同颜色代表看板上的不同栏，横轴每一格表示一天，纵轴显示每天每一栏内都有多少项。从理论上讲，如果工作流程出现障碍，这张图应该可以显示瓶颈的历史变化，以及在制品数量增加与较长的周期时间有什么相关性。

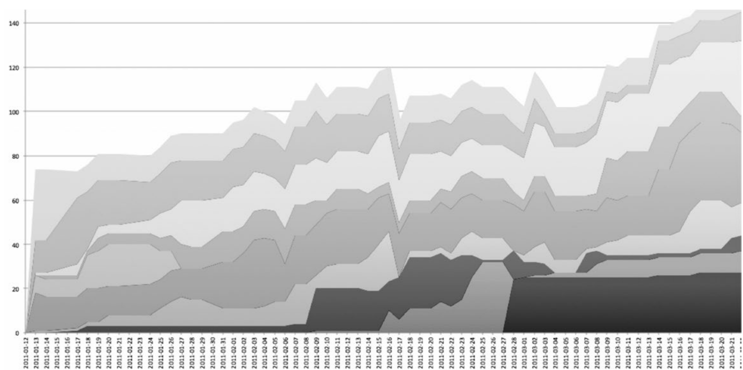
理论上看，这个工具非常棒。

> “理论上看，理论与实践是相同的。实践上看，两者则有不同。”
>
> ——尤吉·贝拉（Yogi Berra）

实践上看，该工具并不适合我们。我们的累计流量图见下页。

从这幅图中很难得出什么有用的结论。即使得出了结论，也有可能是错误的。例如，时间轴的中段好像显示了我们的工作突然大量减少，但实际上是因为我们决定不再将技术故事统计在内。有些情况下，一些功能卡

被我们移放到看板边缘，因为这些功能的开发工作暂停了，所以每天统计看板故事卡的人员就未将其统计在内。等到开发工作恢复后，这些故事卡就又出现在看板内。



事实表明，我们的累计流量图脆弱不堪，只要我们对看板结构稍作改变，或稍微偏离标准工作流，累计流量图就会不准确，且具有误导性。

不过，我们仍然在尽职尽责地收集这一数据，主要是因为我总听到其他教练和精益人士在说累计流量图非常有用。每天收集数据只需要花几分钟时间，所以何乐不为？说不定哪天会用得着……

12.5　流程周期效率

我们没有衡量流程周期效率，不过我在这里专门提到这一点，是因为我们曾想要衡量它。如果我们的看板有电子镜像版，肯定就会衡量流程周期效率的。但现在纯手工的看板系统要做到这一点难度太高。

无论如何，流程周期效率具有如下意义。

$$\text{流程时间效率（\%）} = \frac{\text{接触时间}}{\text{经过时间}}$$

经过时间（Elapsed Time）是指一项功能从项目进度看板的左边抵达右边所需的时间（=周期时间）。

接触时间（Touch Time）是指实际开发测试（或“接触”）一项功能所用的时间。换句话说，就是该功能卡处于在制品栏的总天数，而不是处于排队栏/缓冲池栏的时间。

这会给我们一些很有意思的数据，比如会发现：“嗨，X 功能的开发测试只用了两天时间，但却在项目看板上待了 20 天才下来！流程周期效率只有 10%啊！”

大多数公司都在 10%~15%的范围内,除非他们专门为此优化了工作流。尝试提高这一数字是一个用来弥补并消除浪费的很棒的方式。

第13章

Sprint 与版本发布规划

Sprint 规划会议的目的是弄清楚下一步做什么。就我们的项目而言，这意味着确定哪些功能要进入“下十个功能”栏。

我们的会议并非照本宣科的 Scrum 式 Sprint 规划会议。在 Scrum 模式下，团队需要承诺下一个 Sprint 完成一组特定的功能。我们不这样做。我们甚至没有 Sprint。我们需要的是就接下来要开发的功能达成一致意见。我们的速率还不够稳定，不足以用来预测短期内可以完成多少项功能。



会议内容分两部分：梳理需求清单与挑选前十个功能。

13.1 需求清单梳理

需求清单梳理就是挑选可以转入“可供开发”状态的功能卡（参见第 7 章）。

这项工作是 Sprint 规划会议前半部分的内容，由需求团队向大家介绍接下来需要开发的功能（对，需求团队相当于 Scrum 模式下的产品负责人，主要是因为他们与客户和用户的联系最密切）。

然后我们分成几个跨职能小组，通常每个小组各有一位需求分析师、一位开发人员和一位测试人员。每个小组拿几张功能卡，用规划扑克进行估算（参见第 19 章），并在功能卡上写下小、中或大。如果某张功能卡是“大”，则做进一步细分，或决定暂时将其排除在“下十个功能”范围之外（我们不允许“大”功能进入开发阶段）。他们还会讨论出适当的验收测试方案，并写在卡片背面。

13.2 挑选前十个功能

现在我们手上有一堆功能卡，需要讨论哪些功能应当进入项目看板的“下十个功能”栏（更重要的是，哪些不能进入该栏）。通常开始时的“下十个功能”栏并不是空白，可能已经有两三张功能卡在里面。这种情况下，我们会对照新卡评估已有的卡。讨论的主题是：“在我们接下来需要专注开发的功能里，哪些是最重要的十个功能？”

影响这一决定的因素如下。

- 商业价值——客户最乐于看到的功能有哪些？
- 知识——哪些功能会产生知识（因此会降低风险）？
- 资源利用——我们需要平衡功能领域，这样才能让所有团队都有事做。
- 依赖关系——哪些功能最好组合在一起开发？
- 可测试性——哪些功能放在一起测试最合理，因而需要同期开发？

13.3 为何将需求清单梳理工作移出 Sprint 规划会议

几次 Sprint 规划会议后，我们注意到梳理需求清单非常耗时，导致 Sprint

规划会议总是匆忙结束。所以，我们需要严格控制会议时间，重点讨论主要问题。

于是，我们最近开始单独进行需求清单的梳理，然后再开 Sprint 规划会议。通常在 Sprint 规划会议之前的几天，需求分析团队的一位分析师会与一位开发人员及一位测试人员就接下来的某项功能进行一次非正式的讨论。然后就会对该项功能进行细分、估算，并制定验收测试方案。

我们还是会在 Sprint 规划会议上梳理一些需求，不过通常是在 Sprint 规划会议之前就尽量完成需求清单的梳理工作。我在很多其他组织中也看到了这一趋势。

13.4 规划版本发布

我们很清楚自己的速率：以前平均每周完成三个功能，现在则可以完成四到五个。这个数据对长期的版本发布规划非常有用。我们不知道未来的精确速率，但基本可以肯定每周能完成三至五个功能。

所以，如果有人（挥着功能清单）问大家“年前能完成多少功能”，我们就能给出一个比较切合实际的答案——只要有功能的总数。

问题是，对于长期规划而言，我们并不一定知道功能总数是多少。我们有的只是一堆模糊的创意。我们可以暂且将其称为综述（epic）或功能领域（feature area）。其中有些创意的规模可能非常非常大！

正如 12.2 节所述，我们并不按故事点来估算功能的大小，因为实践表明功能大小的分布其实相当均匀，所以故事点并不会增加多少价值。但对长期规划而言，这一逻辑则行不通，因为我们看的主要是综述，而非具体功能。虽然我们每周完成三至五个功能有可能就完成了一个综述，但要期望我们每周完成三至五个综述则是不现实的！

解决方案相当简单！单独讨论每个综述并估算应将其划分为多少个功

能。这种估算工作（跟其他所有估算工作一样）需要需求分析师、开发人员和测试人员都投入一定的时间和精力。估算过程类似于估算故事点，不过我们讨论的问题是“这一综述有多少个功能”，而不是“有多少个故事点”。

一旦估算好每个综述的具体功能数，我们就可以统计功能总数，然后按照过去每周三至五个功能的速率进行划分。这一做法提供了足够多的信息，能够让我们很有信心地说：“我们应该能在六到十二个月内全部完成。”虽然还只是大致的估算，但却是基于真实数据的。

随着速率趋于稳定，我们可以更好地作出预测，所以我们的答案就可能更精确：八至十个月（只要我们的团队规模不会有太大变动）。

这种规划方式还没有成为企业文化的一部分；我们还是有可能会退回到更为传统的方式，“按功能估算开发时间，然后再汇总”，传统做法很耗时，而且最终得出的计划可能不可行（因为不是基于速率之类的经验性数据）。但至少我们在朝着新的方向努力。

我们付出高昂代价换来的一大经验教训就是，如果没有给力的版本控制系统，要在多团队项目中做好版本发布规划几乎不可能。详细内容请参见下一章。

第14章

我们如何做版本控制

我们一直在多团队情形下以相当快的速度开发一个复杂系统，所以就面临诸多挑战。其中一条经验教训就是，在团队从 30 人扩展到 60 人前，我们就应当先将版本控制系统搞定。很长一段时间以来，我们的版本控制系统都有严重问题，主干与分支都是断裂的。实际上，我们曾有一段时间甚至单独设置了一块看板，专门追踪主干上的所有问题！



为避免今后出现类似问题，我们决定实施主线模型，即我在文章“多团队下的敏捷版本控制”（Agile Version Control with Multiple Teams[1]）中写

① http://www.infoq.com/articles/agile-version-control

到过的稳定主干模式。实施这一变更的过程相当颠簸，但确信无疑的一点是总算让我们回到了正轨！

我们的做法如下。

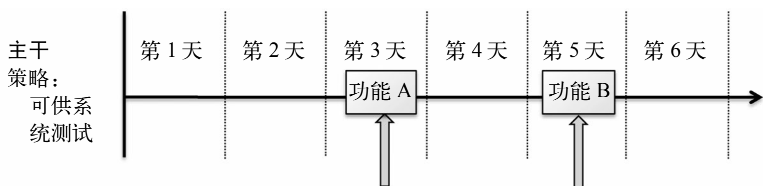
14.1 主干无垃圾

对于这种规模的项目，主干始终稳定至关重要。在我们的项目中，稳定就意味着可以进行系统测试。这与项目进度看板上的“可供系统测试”栏以及写在该栏上的定义直接对应（参见 7.2 节）。



一个功能的代码部分完成后，将代码检入主干并把功能卡拉入“可供系统测试”栏之前，我们会先从功能层次对其进行彻底测试。只有通过所有功能层次的测试，我们才会将代码检入主干并移动功能卡。

这就意味着，从主干的角度而言，产品以谨慎的步骤逐渐递增的同时，主干始终保持稳定，且随时可供系统测试。

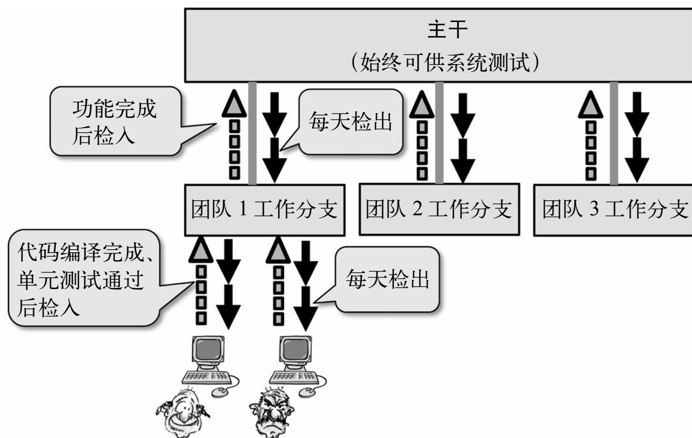


不过，人为错误在所难免，我们也会犯错，也会检入破坏主干的代码。不过没关系，只要错误能被及时发现并得到修复（或回滚）即可。我们的持续集成系统始终监视着主干，任何时候检入了代码都会立即进行版本编译和测试。如果出现错误，熔岩灯就会变色警告我们。持续集成系统不可能验证一切，但可以捕捉到明显的问题，如编译版本出错或单元测试失败。

14.2 团队分支

每个团队都有自己的团队分支，这样他们可以在开发期间检入并共用代码（有些团队有几种不同类型的团队分支）。团队分支的政策比主干宽松得多。代码必须编译，所有单元测试都必须通过，但功能不一定非要完成或测过。采用团队分支的目的是向开发人员提供一个检入未完成代码的地方。

那么，我们如何保持主干、团队分支和所有单个工作站都同步呢？下图总结了我们的变更流程。



基本上，变更连续不断“向下”检出（从主干到团队分支，再到工作站），而“向上”检入（从工作站到团队分支，再到主干）则在固定的点进行。

每天早晨，团队主管将主干上的所有变更与团队分支合并，立即处理所有合并冲突。同样，每位开发人员也要每天将团队分支上的所有变更与他们各自的工作站合并。

只要开发人员认为自己的代码已经很稳定（即，代码已编译，单元测试已通过），就可以将代码检入团队分支，与团队其他成员共享。

只要团队认为他们已经完成一个功能，并做了周到的测试，就可以采取下列行动：

- 检出主干代码并与团队分支合并（以防万一当天有其他团队已更新过主干代码）；
- 最后一次检查代码，确保团队分支代码已经稳定（即可供系统测试）；
- 将团队分支合并到主干。

此时此刻，团队分支与主干代码完全相同！这一光荣时刻很短暂，到下一位开发人员向团队分支检入新代码时即告结束。

这种模式的优点在于，在保持主干稳定的同时，还提供了一个迅速但一致的方式将变更扩散至整个组织。出现在主干上的任何新代码都会在同一天内出现在所有团队分支和所有开发人员的工作站上，所以任何合并冲突都能得以迅速解决。

未合并分支（或分散代码）是一种技术债。在转向稳定主干模型之前，我们曾处于严重拖欠债务的状况。

14.3 系统测试分支

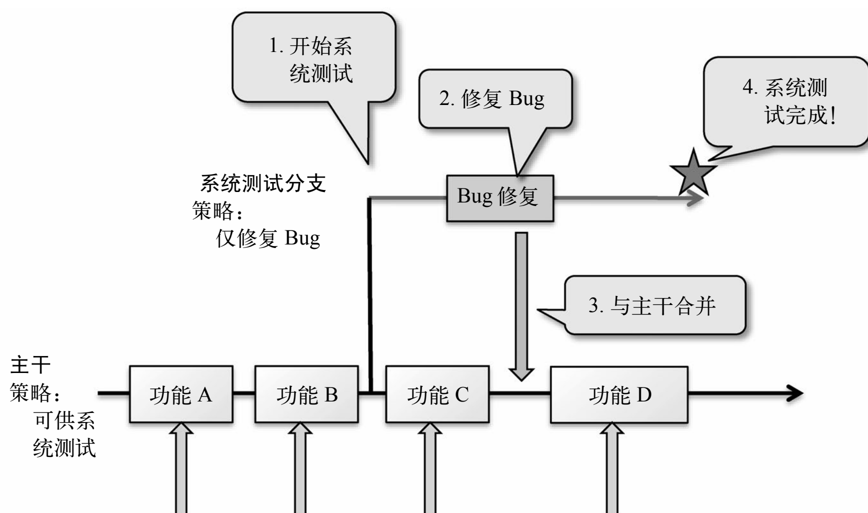
系统测试或多或少都是持续进行的（而不是只在版本发布周期末期进

行）。既然系统测试是测试不同功能组合成整体后的运行情况，我们就需要一个稳定的系统版本来运行测试。也就是说，除 Bug 修复外，我们不希望在进行系统测试期间添加和更新任何功能。

我们运用系统测试分支对此进行管理。只要准备好开始新一轮的系统测试，我们就会在主干基础上创建一个系统测试分支。因为主干始终处于可随时开始系统测试的状态（我们犯错的时候除外），所以测试团队可以立即开始创建工作。脏数据都在团队分支上，所以没有必要等待。

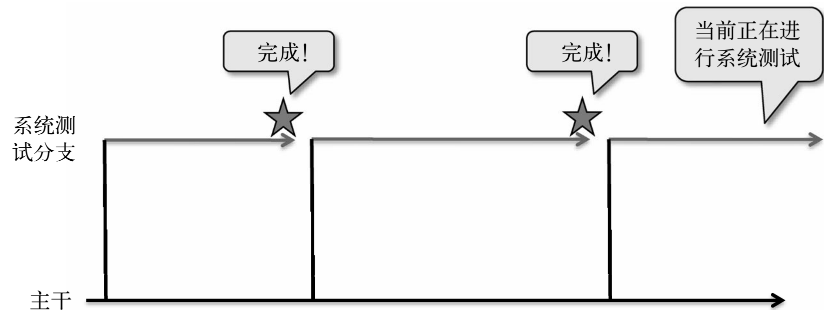
一旦创建好系统测试分支，我们就将该版本部署到系统测试环境中，并开始运行系统测试。所有功能在合并到主干之前就已运行过自动化回归测试脚本，所以在系统测试期间，我们主要做手动情景测试和探索性测试，捕捉较小的 Bug。这个用来测试的系统版本不受主干上任何变更的影响，就等于为测试人员提供了一个稳定的测试版本。

如果系统测试发现了 Bug，开发人员就直接在系统测试分支上修复 Bug，然后直接将修复合并到主干上。这样，Bug 修复能够快速到达所有团队分支上，而且能够肯定该修复会被纳入未来的发布版本中。



测试版本上找不到系统级别的问题时，系统测试工作即告完成。我们可能还缺主要功能，所以还没有到可供验收测试阶段，但目前已经实施好的功能都运行良好。很棒！

那么现在该做什么？当然是再次从头开始！系统测试需要数天才能完成，有时甚至需要数周时间。在此期间，还有团队在往主干上添加新的功能。所以，我们会在主干的最新版本基础上创建一个新的系统测试分支，将其部署到系统测试环境中，然后运行一系列新的系统测试，如此循环往复。



这种工作方式确保我们始终都有一个可以交付验收测试的稳定版本。系统测试中若发现任何问题，我们都能够快速解决，因为自上一次系统测试以来，代码变更很有限。如此下来，每个系统测试周期都会越来越短。

我们并不是每次都运行所有的系统测试。有时会选择一组子测试，具体取决于系统修改了哪些部分，以及我们离版本发布日期还有多长时间。这种风险权衡是针对每个新的系统测试周期单独作出的。

版本控制系统是多团队开发项目的真正核心。随着组织越来越精益、敏捷，版本控制系统也需要与时俱进。所以，对此要时刻关注。要搞清楚从改动一行代码到上线使用需要花多长时间，这可能是项目中最最重要的度量指标！

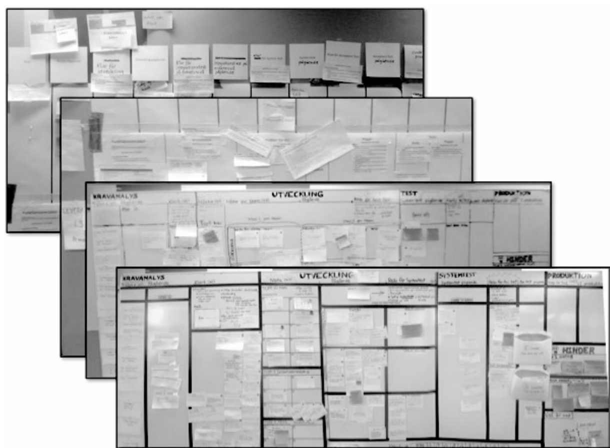
第15章

为何我们只用真实看板

那么，既然有那么多新颖灵活的电子工具可以选择，为什么我们还要使用传统、笨重、凌乱的真实看板呢？还不厌其烦地使用胶带、便利贴，还要手写文字。多数电子工具都能自动生成各种详细的统计数据、进行备份，还可以远程访问、生成不同的视图，等等，为什么不用这些先进的电子工具呢？

一个主要原因就是看板可以演变。

我们的看板版面结构变更过多次，用了两三个月的时间才稳定下来。版面稳定下来以后，我们才开始用黑色胶带来分栏——之前都是手工画分割线，因为变化太频繁。不过，如果有必要的话，我们现在还是可以移动胶带的。



下面列举的是我们经历过的一些变更。

- ❑ 添加或删除栏。
- ❑ 添加或删除平行子栏，有时是在一栏内，有时是在整个看板上。
- ❑ 在看板上添加新标识（索引卡、便利贴、磁贴、彩色胶带等）。
- ❑ 写下策略说明，如“完成的定义”。
- ❑ 写下度量，如速率。
- ❑ 添加彩色标示，如“红色文字表示缺陷”或“粉色便利贴表示障碍”。
- ❑ 用信封把一起发布的所有功能卡装在一起，并在信封上写下版本号。
- ❑ 允许将某些标识放在两栏中间，表示两栏共有。
- ❑ 将大一点的功能切分成多个小功能，并在每张小功能卡上写下关键字，标示所属关系。

每一项变更实施起来都很简单。只要我们清楚想要怎么改，任何人，甚至是五岁小孩，都能够轻松地在看板上完成变更。

除 Google Drawing（谷歌绘图）这样的普通绘图程序外，我还没发现有哪个电子工具能做到所有这些。如果我们再加上一条规则，需要让任何人都能在未经培训的情况下在十五分钟内完成变更，那么，真实看板就真的所向无敌了。

我们曾经根据下面这张草图重新设计了整个看板版面。



根据草图重新做好看板用了大概一个小时的时间。再强调一次，据我所知，大多数电子工具都做不到这一点，而能够做到的电子工具都需要我们具备专业知识才可使用。

我们使用真实看板的第二个原因是需要协作（Collaboration）。

如果没有真实看板，我在第 3 章中描述的“每日鸡尾酒会”就很难进行。



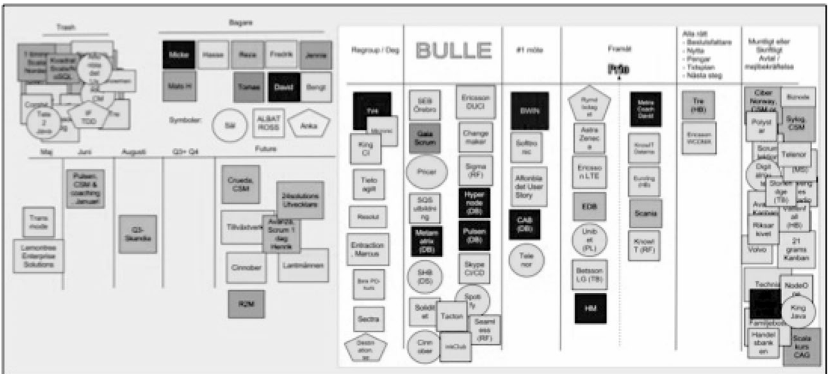
如果采用电子看板，我们可以用投影仪将看板投影到墙上。但这样就会缺乏互动，即大家无法从墙上拿下卡片边讲边示意，无法在卡片上写下文字，也不能在立会期间将卡片移来移去。大家可能要坐到自己座位上以后才会更新看板——这样虽然便利，却没有多少协作性。

就我所服务过的所有客户而言，我注意到的一个明显模式就是，这种看板有助于改变一个组织的企业文化，在 PUST 项目中，这一点确定无疑。我能够看到互动模式的变化，以及团队之间的信任是如何通过每天在看板前的协作而提升的。在第一次项目级的总结会上，我们总结出一些应当继续保持的好做法，其中就有“继续使用看板，以可视化方式呈现所有工作”。

用谷歌绘图应用绘制电子“墙”

在我自己的公司 Crisp 中,我们需要用看板来追踪咨询项目和销售线索。我们很少能聚在一个办公室内，所以真实看板行不通。我们需要电子形式的看板，但又不想牺牲“墙”的灵活性。

实践证明，谷歌绘图（Google Drawing）是非常棒的解决方案！这个程序就是一块画布，就像一块真实的白板或墙壁，你可以在上面画线条、图形，写下文本，还可以拖拽对象，没有任何限制。而且它是在云上的，所以人人都能够同时看到并更新看板，以及跟踪进度。



如果都在同一处办公，我们肯定会用真实看板。不过，对于分布式组织来说，谷歌绘图是我们目前所能找到的最接近“墙”的工具。

我们的确采用了几个电子跟踪系统，作为真实看板的补充，比如 Bug 跟踪系统和用来做版本发布规划的各种电子表格。不过项目进度看板是“总看板”。我们一致同意要保证项目看板反映真实情况，并保持实时更新。所含信息与项目看板内容相同的任何其他电子文档都被我们视为看板的“影子”——如果有信息不同步的情况，始终以看板为准。

我们也曾讨论过引入电子工具来复制项目看板上的概要信息。这样我们就可以将部分度量自动化，并制作一个高层概要电子看板，让位于其他地方的管理高层和相关人员都能够看到。保持电子看板和真实看板信息同步可能需要花费一些精力，但可能很值得。这个电子看板跟其他电子工具一样，会成为真实看板的补充，但不是替代它。不过我们还未尝试这一点。

可见，我们的旅程还远远没有抵达终点，总有可以提高改进的地方！

第16章

经验教训

这一章将结束案例研究部分的内容。希望你喜欢这段行程！

你可能已经看出来了，此案例的真正主题是组织变革！下面是从变革推动者角度总结出的几个关键要点（任何人都可以是变革推动者）。

16.1 了解目标

把所有人都召集到一起，让大家就“完善”的定义达成共识。“完善”的流程、组织和工作环境应当是什么样的？这将成为指示你所在组织变革方向的指南针，因此目标不一定非要切实可行。完善是方向，而不是终点！拥有了明确的方向，再专心评估改进工作就会容易得多。

16.2 不断实验

不要试图寻找完美的解决方案。可能压根儿就没有所谓的完美解决方案，就算你找到了，也有可能未能正确领会。正确的做法是，寻找微小的渐进式改进机会，并将其视作*实验机会*。实验有可能实现预期改进，也有可能不会实现，但能让我们从中积累经验教训，然后用于设计新的实验。

伟大的流程不是设计出来的，而是*逐渐演变*出来的。所以，重要的不是流程，而是我们用来*改进*流程的流程。

16.3　拥抱失败

有些变革行不通，有些变革甚至会把工作搞得更糟。不必为此担心，因为失败不可避免。担心失败则是创新的最大敌人。我们应当反思的是“我们学到了什么？下次应当做出什么尝试？”，而不是“我们怎么会失败？是谁把事情搞砸了？”

唯一的真正失败是未能从失败中吸取经验教训。

16.4　解决真正的问题

无论何时尝试作出变革时，都要不断问自己：“我在解决什么问题？这个问题是真实存在的还是假设的？还有没有其他我应当先解决的更重要的问题？”如果不能确定，就要询问别人！我们一不小心就会去钻研些无关问题或假设问题，尤其是作为来自组织外部的短期教练而言。

16.5　拥有专职变革推动者

变革很难，涉及人事的变革更是难上加难，而组织变革总是会涉及人事。一个关键的成功要素是，至少应拥有一名专职的变革推动者，完全专注于推动、领导和协调变革过程。

如果有两位专门的变革推动者则更为理想：一位内部人士，一位外部人士。内部人士（通常是员工）拥有特定领域知识，知道什么工作应该找谁，他了解组织的历史以及以往的成功做法。外部人士（通常是咨询专家）则可以提供全新的视角，他拥有帮助其他公司完成类似变革的成功经验。

文化可以定义为“所有人都遵循但却不会注意到的做法”，内部人员容易“不识庐山真面目”，而外部人士则更有可能注意到现状并提出质疑。

16.6　让人们参与进来

多数人都喜见变革，他们只是不喜欢被别人改变。所以，在做任何改变之前，首先必须让有可能会受到影响的人参与进来。强迫人们去改变通常都不会见效，最终只是多此一举，而且让人很痛苦。如果人们抵制你伟大的变革提议，那就有可能是你没把问题说清楚，要么就是你没找出真正的问题。回去仔细研究因果图（详细内容请参见第 20 章），再认真思考一下！

更好的做法是，先不要自己作出变革提议！而是将你看到的问题用可视化方式呈现出来，让那些相关的人们参与进来，提出他们自己的解决方案。如果是大家自己提出来的意见，人们就会更容易接受变革！

一旦所有人都认识到问题真的是问题，而且需要解决的时候，你就已经事半功倍了！

Part 2 第二部分

技术详解

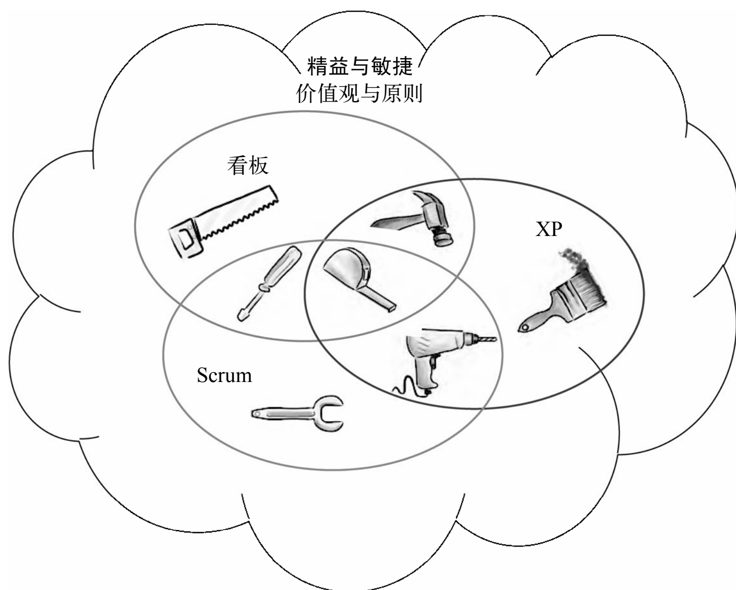
想了解更多内容吗？那就跟我们一起来详细了解本书提到的一些技术吧。

第17章

敏捷与精益概述

多数读者可能已经对敏捷和精益原则有所了解。对于不太熟悉相关技术的读者,本章会对基本概念和原则做简要介绍,并演示这些技术与 Scrum、极限编程（XP）和看板等相关方法有何关联。

广义而言，精益与敏捷是两组具有高度兼容性的价值观和原则，都阐述了如何成功地进行产品开发。Scrum、XP 和看板则是将这些原则运用到实践中的三种具体方法。换句话说，它们是精益和敏捷软件开发里轻度重叠的三种不同风格。



Scrum、XP 和看板都有很具体的技术，如 Sprint 规划会议（Scrum）、结对编程（XP）和限定在制品（看板）。这些技术都可视作流程工具。这三种工具的功能都有相当程度的重叠，例如，三种工具都建议使用真实的任务板将当前工作以可视化方式展现出来。

17.1　敏捷概述

术语敏捷软件开发出现于 2001 年。当时，来自软件开发界的十七位思想领袖聚集在美国犹他州的一个滑雪度假胜地，探讨软件开发如何取得成功。此前，他们都各自创立了不同的新方法，如 Scrum、XP 和动态系统开发方法（DSDM）。研讨会期间，他们总结出一些强大的共同观点，形成了软件开发如何成功的共有愿景，即后来人们熟知的《敏捷宣言》。

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

《敏捷宣言》内容如下。

我们正在通过亲身实践以及帮助他人实践，探寻更好的软件开发方法。通过这项工作，我们建立了如下价值观：

- **个体和互动**胜过流程和工具
- **可以工作的软件**胜过详尽的文档
- **客户合作**胜过合同谈判
- **响应变化**胜过遵循计划

也就是说，虽然右项也具有其价值，但我们认为左项具有更大的价值。

研讨会结束后，他们就支撑这些价值观的以下十二条原则达成共识。

- 我们最重要的目标，是通过持续不断地及早交付有价值的软件使客户满意。
- 欣然面对需求变化，即使在开发后期也一样。为了客户的竞争优势，要通过敏捷过程掌控变化。
- 经常地交付可工作的软件，比如相隔几星期或一两个月就交付，倾向于采取较短的周期。
- 业务人员和开发人员必须相互合作，项目中的每一天都不例外。
- 激发个体的斗志，以他们为核心搭建项目。提供所需的环境和支援，辅以信任，从而达成目标。
- 不论团队内外，传递信息效果最好、效率也最高的方式是面对面的交谈。
- 可工作的软件是进度的首要度量标准。
- 敏捷过程倡导可持续开发。责任人、开发人员和用户要能够共同维持其步调稳定延续。
- 坚持不懈地追求技术卓越和良好设计，敏捷能力由此增强。

- 以简洁为本，它是极力减少不必要工作量的艺术。
- 最好的架构、需求和设计出自自组织团队。
- 团队定期地反思如何能提高成效，并依此调整自身的举止表现。

虽然敏捷一词正式出现于 2001 年，但多数敏捷方法都是在 20 世纪 80 年代到 90 年代形成的。敏捷只是一个描述了共同特征的统称。凡遵循上述价值观和原则的方法或方式都可视为敏捷方法。

《敏捷宣言》是一份呼吁变革的历史文档。宣言中的文字在全球成千上万人心中引起深刻共鸣，引发了软件开发革命。十年后，所有作者（其中一位除外）再次聚集在相同地点，重申他们依旧支持敏捷价值观和原则。因此，尽管软件行业发生了翻天覆地的变化，《敏捷宣言》却经历住了时间的考验。

没有人拥有敏捷软件开发这一术语的所有权，所以它可以有很多种解释。我们通过宣言可以了解 2001 年时的敏捷是什么意思，今天它的定义已经没有那么明确。宣言的最初作者中，很多人希望这个术语会最终淡出历史，标志着敏捷价值观和原则已成为“我们开发软件的方式”。

17.2 精益概述

精益起源于日本丰田公司的“TPS”（丰田生产方式），即助力丰田成为全球最成功汽车制造商的生产方式。实践证明，TPS 的基本原则“丰田之道”几乎适用于所有行业，包括软件开发。

敏捷与精益可以看作是一对拥有共同价值观但起源不同的兄弟。精益起源于制造业，敏捷起源于软件开发。两组原则都能与对方完美契合，而且适用范围都非常广泛。越来越多的软件开发组织在探索如何将两组原则完美结合，从而应用于从产品创意到交付的完整开发链。

帕彭迪克夫妇在将精益原则与软件开发结合方面做出了卓越贡献，以

下是他们精炼出的精益原则[①]。

1. 全局优化

局部的优化长期来说，会对系统整体优化不利。

- 专注于整体价值流：从概念到现金。从客户需求到软件部署。
- 交付完整产品：客户不要软件产品，他们要解决问题。完整的解决方案是由完整的团队构建的。
- 着眼长期：警惕导致短期思维和优化局部业绩的治理和激励体系。

2. 消灭浪费

浪费指所有那些不能增加客户价值的事项。软件开发中的三大浪费如下。

- 构建错误的功能："没有什么比高效完成根本不应做的工作更无用。"
- 拒绝学习：我们有很多策略都干扰了我们学习，例如，只按计划行事、频繁移交、决策与工作分离等，而学习则是开发的精髓。
- 辗转现象：那些干扰价值顺利流动的做法，例如，任务切换、请求清单冗长、大堆未完成的工作等等，都只能达到事倍功半的效果。

3. 品质为先

如果在验证过程中总是能发现缺陷，那流程就有问题。

- 最终验证不应发现缺陷：所有软件开发流程的根本目的都是尽早在开发阶段发现并修复缺陷。
- 采用测试先行的开发模式让流程具有防误机制：测试（包括单元测试、端对端测试和集成测试）必不可少，以此建立信心，保证系统在任何层次、在开发阶段任何时间点都始终正确无误。
- 打破依赖：系统架构应当支持随时添加功能。

① www.poppendieck.com

4. 不断学习

规划工作非常有用。学习则必不可少。

- 可预测的性能来自于反馈：可预测的组织不会猜测未来并称之为计划；反之，他们会培养能力对未来做出响应。
- 保持选择方案：视代码为实验——使其具有容变性。
- 最后可靠时刻：在做出不可逆转的决策之前尽可能学习。不要提前做出纠正代价高昂的决策，也不要事后才做出决定！

5. 尽快交付

从一开始就深入了解所有干系人看重的价值。然后基于这样深入了解的价值观，创建稳定、连贯的工作流。

- 快速交付、高质量和低成本是完全相互兼容的：以速度竞争见长的公司拥有很高的成本优势，他们可以交付优质的产品，而且对客户需求更为敏感。
- 排队理论同样适用于开发，而不仅仅是服务行业：专注于使用性会造成交通堵塞，反而降低了使用性。以较小的批量、限制同时进行的工作数来缩短周期时间。大力限制等待清单和队列的长度。
- 管理工作流比管理进度表要容易得多：建立可靠、可预测交付物的最佳方式是通过迭代和看板建立可靠、可重复的工作流。

6. 人人参与

聪明、有创造力的人员的时间与精力，是当代经济的稀有资源和竞争优势的基础。

获得公正薪资的人员在自主性、成长性和使命感等方面受到激励[1]。

- 自主性：最有效的工作小组是半自治团队，有一个内部主管从头到尾负责完整、有意义的任务。

---

① http://www.youtube.com/watch?v=u6XAPnuFjJc

- 成长性：对人员的尊重意味着提供挑战、反馈和让所有人都能够发挥潜能、表现卓越的良好环境。
- 使命感：将工作与价值挂钩。只有相信自己工作的意义，团队成员才会全心投入工作，实现这种使命。

7. 不断提高

结果不是重点——重点是培养人、发展体制，使之能够交付结果。

- 失败是个学习机会：即使是非常小的失败都会被深入调查并纠正的，做到一丝不苟的时候，才可能获得最可靠的性能。
- 标准存在的目的就是要被质疑和提高的：将现行的、最知名的做法纳入人人都遵循的标准，与此同时鼓励所有人质疑并改变标准。
- 使用科学方法：教团队建立假设、开展大量快速实验、创建简明文档并实施最佳方案。

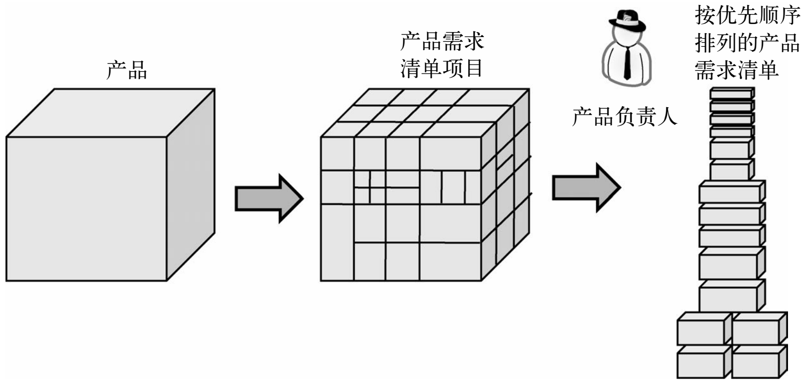
17.3 Scrum 概述

Scrum 是由杰夫·萨瑟兰（Jeff Sutherland）和肯·施瓦伯（Ken Schwaber）于 20 世纪 90 年代早期共同创建的一种软件开发过程。Scrum 根植于经验过程控制和复杂自适应系统理论，受《哈佛商业评论》1986 年一篇题为“新新产品开发游戏”（New New Product Development Game）的文章的启发而来。

Scrum 的核心概念如下文所述。

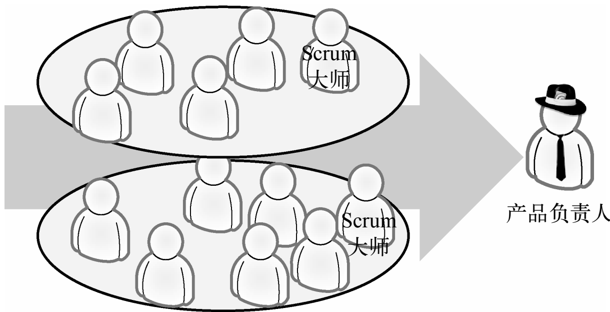
1. 按优先顺序排列的产品需求清单

将产品分割成一组小而具体的可交付物，即产品需求清单。产品负责人对产品愿景进行定义，并按商业价值以及风险和依赖关系等其他因素对需求清单进行排序。



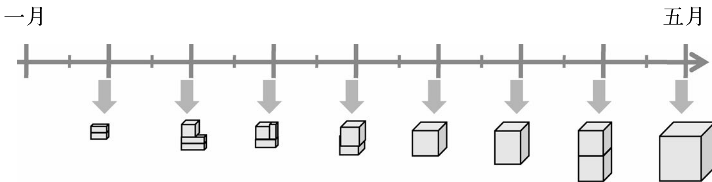
2. 跨职能团队

将产品所有人员划分为多个小规模、跨职能、自组织的开发团队。每个团队都有一位产品负责人负责定义愿景和总体的业务优先顺序，以及一位 Scrum 大师专注于改进团队、消除障碍。



3. Sprint

将整个开发时间划分成多个短小的、固定的迭代周期或 Sprint（通常为两周或三周）。开发团队自行决定每个迭代周期要完成多少个产品需求清单项。每个迭代周期最后都要演示已通过测试、能够发布的版本。



4. 持续调整版本发布计划

产品负责人与客户一起合作，在每个迭代周期之后仔细检查发布版本，根据所得的结果，不断优化版本发布计划，并更新优先排序。

5. 持续调整流程

开发团队通过每个迭代周期之后的回顾会议不断优化开发流程。

所以，Scrum 开发模式意味着：

不是由一个大团队用很长的时间来开发一个大产品……

……而是由一个小团队用很短的时间来开发一个小功能。

但定期集成，以构成整体。

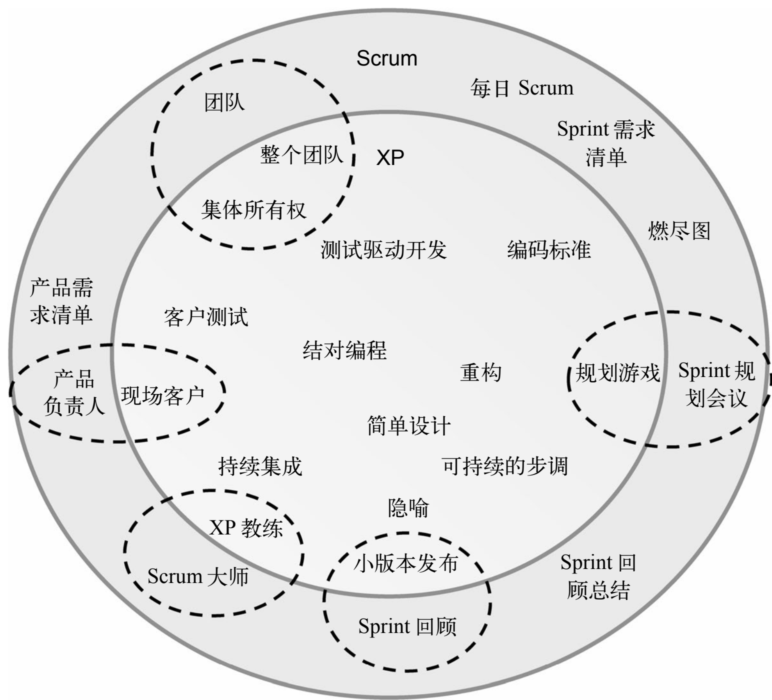
Scrum 模式不会硬性规定任何具体的工程实践——这些都由团队自行决定。不过，在实践中，不纳入 XP 的核心工程实践而通过 Scrum 模式取得成功是非常困难的。

17.4 XP 概述

极限编程（XP）是肯特·贝克于 20 世纪 90 年代中期创立的软件开发方法。该方法以简洁、沟通、反馈、勇气和尊重等价值观为基础。XP 方法是与 Scrum 并行发展的，实际上包含了大多数相同要素。例如，XP 中的现场客户就大致等同于 Scrum 中的产品负责人。

从这个意义上而言，Scrum 可被视作 XP 的“包装纸”，专注于结构问题和外部沟通。而 XP 除多数理念都与 Scrum 相同以外，还增加了一些团队内部的工程实践，包括以下内容。

- 持续集成：拥有一个随着团队的开发可自动编译、集成并测试代码的系统。这样就能尽早为开发团队提供有关产品质量方面的反馈。
- 结对编程：在一台工作站上进行结对编程，从而使学习效果最大化、设计质量最大化、缺陷最小化。



- **测试驱动开发**：采用测试代码驱动系统的设计。编写自动化测试脚本，然后编写刚刚足够的代码以使其通过测试，然后从根本上重构代码，提高其可读性，移除重复代码。清理并重复这一过程。
- **集体代码所有权**：允许（实际上是鼓励）开发团队的任何人编辑代码库的任何部分。这样可营造出团队所有权的意识，确保整个系统的设计都一致、易于理解。
- **增量式设计改进**：从最简单的设计开始，然后运用重构等技术持续不断地改进设计，而不是从一开始就做好完整的设计。

上述许多实践都互为基础。例如，如果系统的自动化测试覆盖范围不足，那增量式设计改进就很难实现、令人生畏且风险很高，而若要测试覆盖范围足够，则需要通过测试驱动开发和结对编程才可实现。不过，如果所有的测试都必须手动触发，而且只能在开发人员的本地工作站上运行，

问题就会让人更头痛，所以，我们就需要一个持续集成系统在后台自动完成上述工作，等等。

17.5 看板概述

看板是敏捷软件开发的精益方法。

实际上，看板有着多方面的意义。从字面上看，看板是日语单词，是“可视卡片”（或标志）的意思。在丰田，看板专指将整个精益生产系统连接在一起的可视化物理信号系统。

不过，大卫·安德森于 2004 年率先提出了软件开发的精益思维实施方法和约束理论[①]。在唐·赖纳特森等专家的指导下，演变成为大卫所称的“软件开发看板系统”，现在人们都将其简称为“看板”。

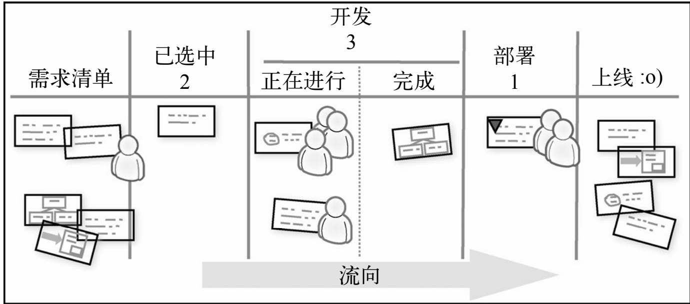
所以，虽然看板应用于软件开发还相当新颖，在精益生产中却已有半个多世纪的历史了。

看板的规则很简单。不过，跟象棋一样，规则简单并不意味着游戏简单。

- 可视化工作流：
  - 把产品切分成小块，将每一块写在一张卡片上，然后将卡片贴到墙上；
  - 墙上的每一栏都有名称，以此显示每张卡片在工作流中所处的位置。
- 限定在制品（WIP）：针对工作流的每个状态，明确限定正在进行中的工作项数量。
- 衡量并管理周期时间：完成一个工作项的平均时间，有时称为前置时间（更贴切的术语可能应该是流通时间）。优化流程，让周期时间尽可能短、尽可能可预测。

---

① http://en.wikipedia.org/wiki/Theory_of_Constraints

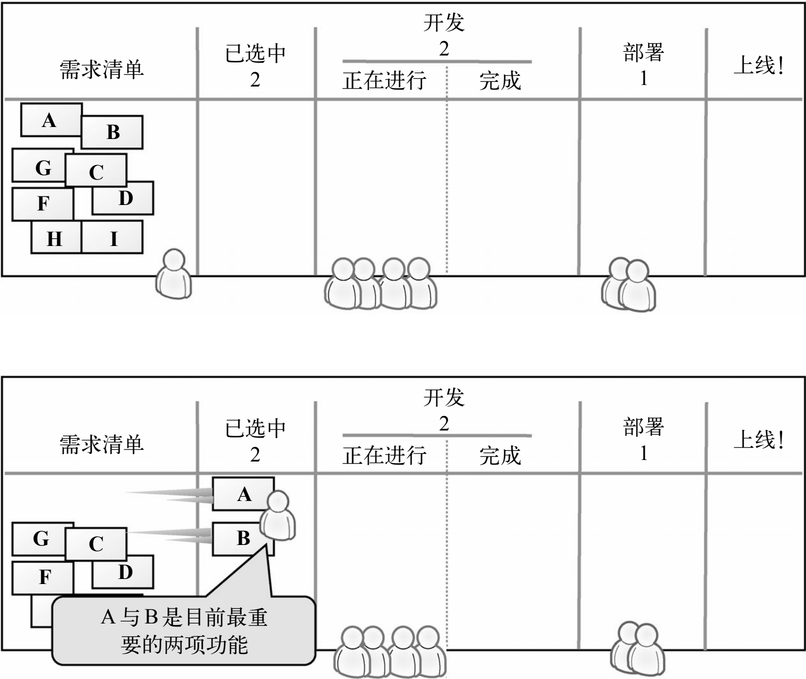


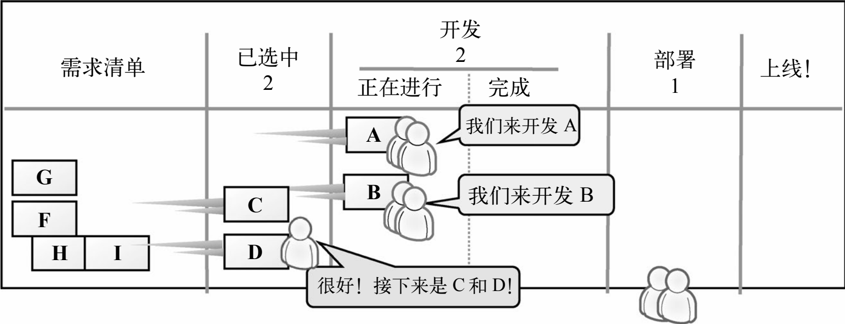
这就基本上直接实施了精益拉式生产调度系统。

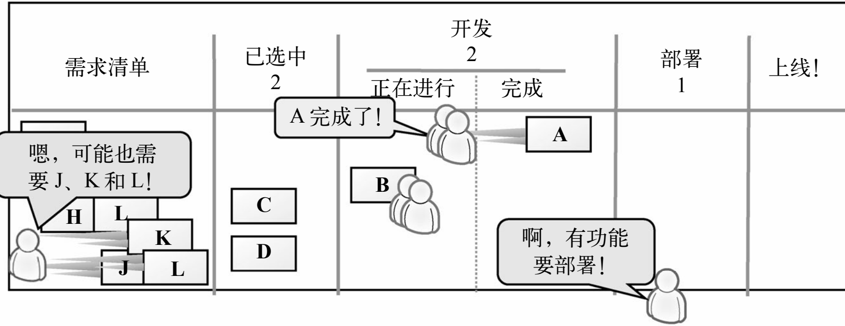
Scrum 专注于结构和沟通，XP 增加了工程实践，看板则专注于将工作流可视化，并对瓶颈进行管理。

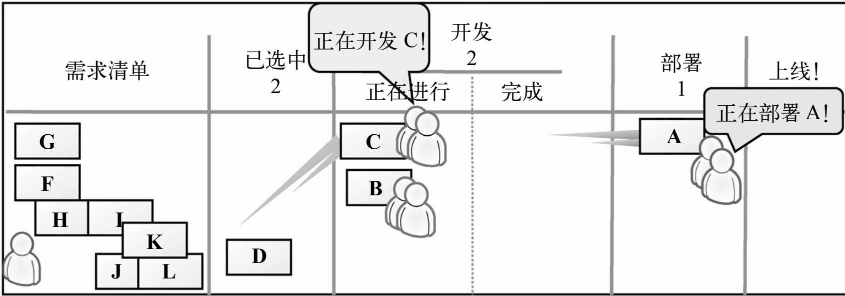
看板管理的典型一天

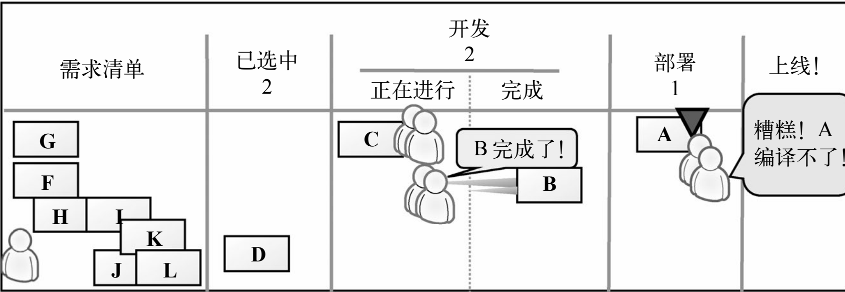
下面这几幅图形象地展示了看板管理试图推动的行为类型。

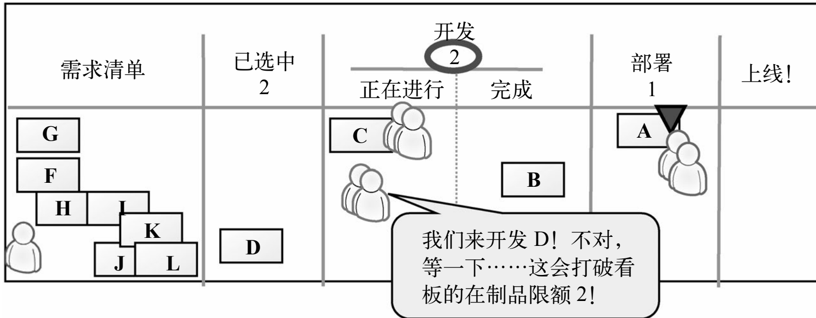


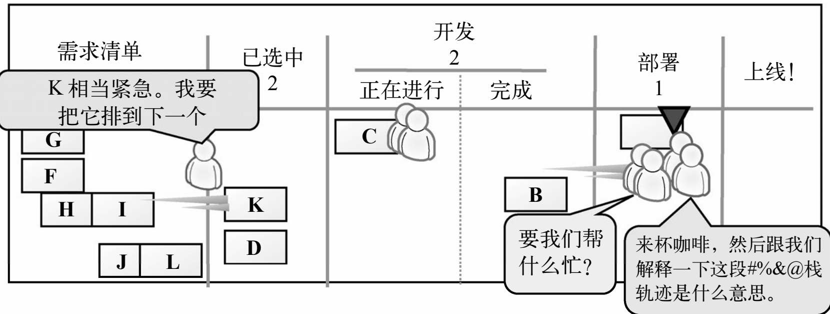

需求清单
已选中
2
开发
2
正在进行
完成
部署
1
上线!
A
B
C
D
G
F
H
I
我们来开发 A
我们来开发 B
很好！接下来是 C 和 D!

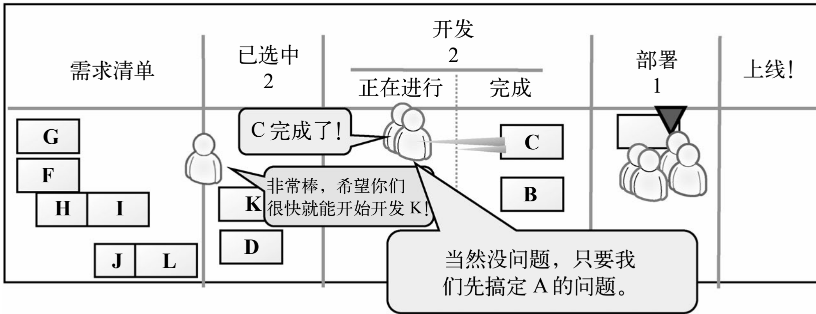

需求清单
已选中
2
开发
2
正在进行
完成
部署
1
上线!
A 完成了!
A
B
C
D
H
L
K
J
L
嗯，可能也需要 J、K 和 L!
啊，有功能要部署!

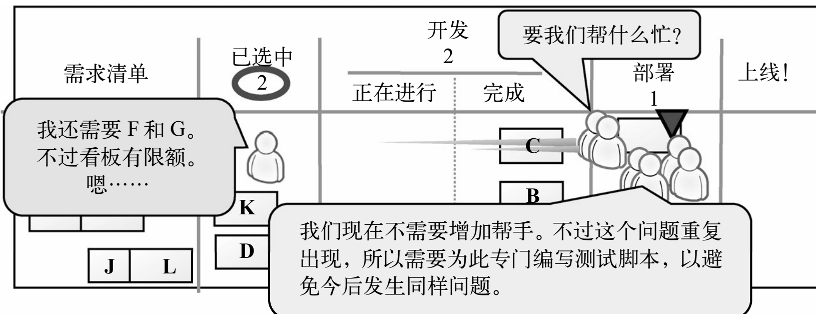

需求清单
已选中
2
开发
2
正在进行
完成
部署
1
上线!
正在开发 C!
正在部署 A!
G
F
H
I
K
J
L
C
B
D
A

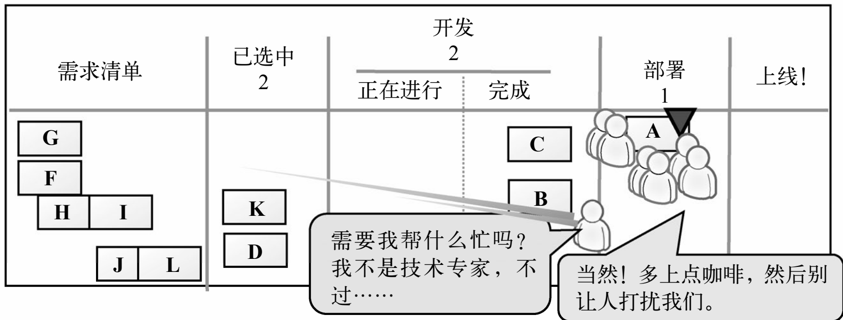

需求清单
已选中
2
开发
2
正在进行
完成
部署
1
上线!
G
F
H
I
K
J
L
C
D
B 完成了!
B
A
糟糕！A 编译不了!


需求清单
已选中
2
开发
2
正在进行
完成
部署
1
上线!
G
F
H
I
K
J
L
D
C
B
A
我们来开发 D！不对，等一下……这会打破看板的在制品限额 2！

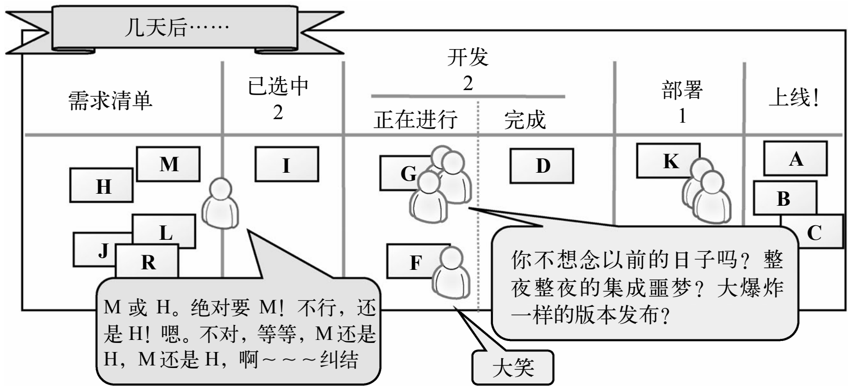

需求清单
已选中
2
开发
2
正在进行
完成
部署
1
上线!
K 相当紧急。我要把它排到下一个
G
F
H
I
K
D
J
L
C
B
要我们帮什么忙？
来杯咖啡，然后跟我们解释一下这段#%&@栈轨迹是什么意思。


需求清单
已选中
2
开发
2
正在进行
完成
部署
1
上线!
G
F
H
I
J
L
K
D
C 完成了！
非常棒，希望你们很快就能开始开发 K！
C
B
当然没问题，只要我们先搞定 A 的问题。


需求清单
已选中
2
开发
2
正在进行
完成
部署
1
上线!
要我们帮什么忙？
我还需要 F 和 G。不过看板有限额。嗯……
K
D
J
L
C
B
我们现在不需要增加帮手。不过这个问题重复出现，所以需要为此专门编写测试脚本，以避免今后发生同样问题。



需求清单
已选中
2
开发
2
正在进行
完成
部署
1
上线!
G
F
H
I
J
L
K
D
C
B
A
需要我帮什么忙吗？我不是技术专家，不过……
当然！多上点咖啡，然后别让人打扰我们。



几天后……
需求清单
已选中
2
开发
2
正在进行
完成
部署
1
上线!
M
H
L
J
R
I
G
F
D
K
A
B
C
M 或 H。绝对要 M！不行，还是 H！嗯。不对，等等，M 还是 H，M 还是 H，啊～～～纠结
大笑
你不想念以前的日子吗？整夜整夜的集成噩梦？大爆炸一样的版本发布？

第18章

缩减测试自动化需求清单

很多有遗留代码库的公司在转向敏捷的时候都遇到一大障碍：测试未实现自动化！

测试未实现自动化的情况下，要在系统中做出改动就会非常困难，因为一旦发生问题，也不会有人注意到。等到新版本正式上线，真正的用户发现缺陷时，局面就会非常尴尬，而且修复起来代价高昂——更糟的结果是导致一系列修复，因为每次修复都会引入让人预料不到的新缺陷。

这就让开发团队非常害怕改动代码，因此就不情愿改进代码设计。于是，随着开发工作的推进，代码质量呈螺旋形下降。

好消息是你可以改善这种情况！

18.1　怎么办

在这种情况下，你有以下一些选择方案。

- 忽略问题。让产品衰败而亡，并希望到那时候没有人再需要使用该产品。
- 采用测试驱动开发模式，从零开始重新构建系统，保证测试覆盖率达标。
- 单独启动一个测试自动化项目，由专门的团队提高产品的测试覆盖率，直到达标。

- 让团队在每个迭代周期都逐渐提高测试覆盖率。

猜猜哪种方法效果最好？对，最后一个——每个迭代周期逐渐提高测试覆盖率。至少我的经验是这样。

第三个方案听上去很诱人，但却有风险。谁来具体实施测试自动化？单独的团队吗？如果是这样，是否就意味着其他开发人员不需要了解如何运行自动化测试？这是个问题。还是由整个团队来实施测试自动化项目？这种情况下，在完成之前，（从业务角度而言）他们的速率就都是 0。那么，他们什么时候做完？什么时候才算测试自动化“结束”？

我们还是来看看第四个方案吧——每个迭代周期逐渐提高测试覆盖率。那么，具体应该如何做呢？

18.2 如何每个迭代周期都提高测试覆盖率

下面是我喜欢用的方法：

(1) 列出测试用例；

(2) 按风险大小、手动测试的成本、自动化测试的成本将测试分类；

(3) 按优先顺序对列表进行排序；

(4) 从优先级最高的开始，每个迭代周期都对若干测试实施自动化。

我们来看看每一步都怎么做。

18.3 第 1 步：列出测试用例

想一想自己今天是如何测试产品的。用头脑风暴的方式列出最重要的测试用例——今天已经手动执行过的或者希望自己有时间执行的。下面是一个假设的在线银行系统示例。

测试用例
改变皮肤
安全警报
交易历史
封存账户
添加新用户
查询结果排序
存入现金
验证转账

18.4 第 2 步：测试分类

首先，按风险大小划分测试用例。看一看测试列表，目前暂时忽略手动测试成本。现在，如果丢弃其中一半的测试，而且永远不会执行这些测试会如何？你会保留哪些测试？这个因素要综合考虑失败的可能性和失败的成本。

标出风险高的测试，就是那些让你晚上睡不着觉的测试。

测试用例	风险
改变皮肤	
安全警报	高
交易历史	
封存账户	高
添加新用户	
查询结果排序	
存入现金	高
验证转账	高

现在想一想，手动执行测试的话，每一项测试都需要多长时间。哪一半的测试花费的时间最长？把这些测试标记出来。

测试用例	风险	手动测试成本
改变皮肤		
安全警报	高	
交易历史		高
封存账户	高	高
添加新用户		
查询结果排序		高
存入现金	高	
验证转账	高	高

最后，估算一下每一项测试编写自动化脚本的工作量。标出工作量最高的那一半。

测试用例	风险	手动测试成本	自动化测试成本
改变皮肤			高
安全警报	高		高
交易历史		高	
封存账户	高	高	
添加新用户			
查询结果排序		高	高
存入现金	高		
验证转账	高	高	高

注意，手动测试成本在每次运行测试的时候都会发生，而自动化成本则是一次性的。所以，编写测试自动化脚本实际上是投资，而不是成本。

18.5 第3步：按优先顺序对列表进行排序

你认为我们应当先把哪项测试自动化？是不是应当先自动化“改变皮肤”？这一项风险低、手动测试易于执行，不过难以自动化。还是应当先

自动化“封存账户”？这一项风险高、手动测试难度大，但易于自动化。这个决定显而易见。

下面这个决定有一点难度了。我们是不是应当自动化“验证转账”？这一项风险高、手动测试难度大，而且自动化难度也大。还是应当自动化“存入现金”？这一项风险也高，但手动测试容易，也易于自动化。这个决定就要根据具体情况来做。

关于哪些测试应当先予以自动化，基本上你需要做的决定有以下三个。

- 风险高但手动测试容易的测试，或者风险低但手动测试难度大的测试。
- 易于手动测试且易于自动化的测试，或者手动测试难度大且自动化难度也大的测试。
- 风险高且难以自动化的测试，或者风险低且易于自动化的测试。

做好这些决定就可以按优先顺序排列测试用例。在这个银行示例中，我决定首先对手动测试成本进行优先排序，然后是风险，再然后是自动化成本。下面就是我排好序的列表。

最先自动化

测试用例	风险	手动测试成本	自动化测试成本
封存账户	高	高	
验证转账	高	高	高
交易历史		高	
查询结果排序		高	高
存入现金	高		
安全警报	高		高
添加新用户			
改变皮肤			高

最后自动化，或不自动化

所以，结果出来了！我们得出了按优先顺序排序的测试自动化故事需求清单。

当然，我们也可以发明一种计算算法。例如，可以给每个标记出的单元格分配一个点，然后只需要把每一行加总和排序。或者可以根据直觉，直接手动对列表进行排序。

还可以对每个类别采用更精确的单位，比如小时和故事点。

测试用例	风险	手动测试成本	自动化测试成本
封存账户	高	5小时	0.5个故事点
验证转账	高	3小时	1个故事点
交易历史	中	3小时	1个故事点
查询结果排序	中	2小时	8个故事点
存入现金	高	1.5小时	1个故事点
安全警报	高	1小时	13个故事点
添加新用户	低	0.5小时	3个故事点
改变皮肤	低	0.5小时	20个故事点

不过要记住，目前的目标只是对列表进行优先排序。如果能用简单自然的分类办法，就没有必要把问题复杂化，对吧？分析很有用，但过度分析就是浪费时间。

无论如何，我们现在排好测试自动化需求清单的优先顺序了！

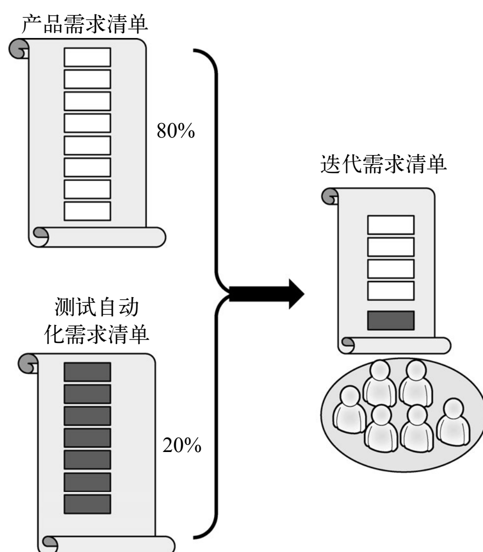
18.6 第 4 步：每个迭代周期自动化若干测试

不论是否有测试自动化需求清单，每个新功能都应包括功能级别的自动化测试。这是 XP 的实践，称为*客户验收测试*。如果不做功能级别的自动化测试，一开始就会让产品陷入一团乱麻。

不过，除了实施新故事外，我们需要花点时间将以前故事的老测试用例自动化。那需要花多长时间呢？开发团队需要与产品负责人具体协商。

例如，我们可以同意让团队将 80%的时间花在开发产品需求清单列举的新功能上，然后把 20%的时间花在测试自动化需求清单上。那么，在每次的

迭代规划会议上，团队就需要同时从两个需求清单中拉入故事卡。



再比如，我们还可以达成以下一些共识。

- 每个迭代周期都要实施一个测试自动化故事。
- 每个迭代周期都会实施若干测试自动化故事，故事点总和不超过 10。
- 每个迭代周期都会先完成产品需求清单的故事，然后若有剩余的时间，再来实施测试自动化故事。
- 产品负责人会将测试自动化故事合并到总体的产品需求清单中，开发团队会对所有故事一视同仁。

这些共识的具体形式是什么样子则不重要。如果有必要，可以每个迭代周期都改变具体形式（只要团队和产品负责人同意）。重点是测试自动化债务得以一步一步逐渐偿还。

等到测试自动化需求清单上的一半故事完成后，你可能会觉得："嗨，现在已经还够债务了！我们干脆跳过其余的老测试用例，它们都不值得做自动化了。"然后你干脆把剩余的都丢弃。如果做到这一步，那我要恭喜你！

18.7　这能解决问题吗

遵循这种模式不会魔法般地解决测试自动化问题，至少不会在短期内解决。不过，这种模式会让问题更容易解决。几个月后你就会注意到显著的不同。

第19章

用规划扑克估算需求清单大小

规划扑克是简单却强大的工具，能让团队以较快的速度完成规划工作（即估算开发一项功能需要的工作量），同时也让规划工作更准确、更好玩。这个术语由詹姆斯·格瑞尼（James Grenning）创造，后由麦克·科恩（Mike Cohn）推广开来。

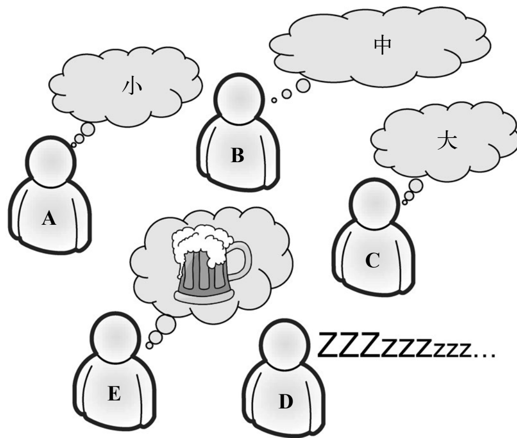


19.1 不用规划扑克进行估算

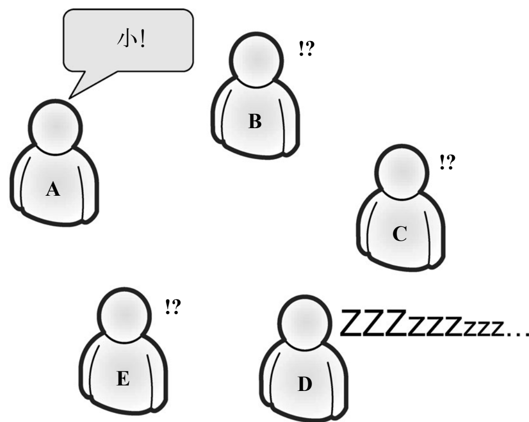
下面是团队在进行估算时遇到的典型问题。假设我们在 Sprint 规划会议上，产品负责人说……



于是，团队成员都开始思考这项功能有多大。

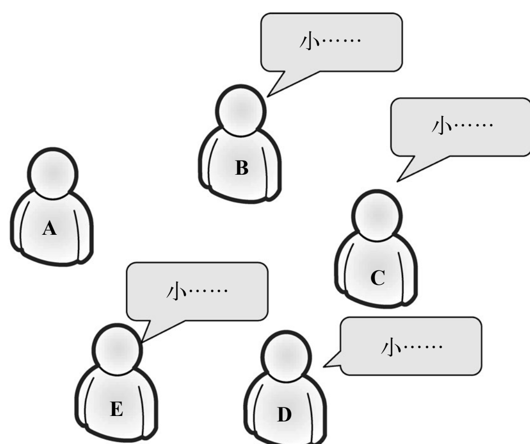


A 先生认为自己非常清楚实现这项功能都需要做什么工作，而且没多少工作要做。B 女士和 C 女士则比较悲观。D 先生和 E 先生在偷懒。于是，A 先生说……



这个意见让B女士和C女士很困惑，她们开始怀疑自己的估算结果。E先生醒过来了，还不知道大家在估算什么。D先生仍在打盹中。

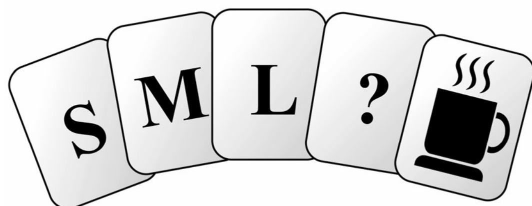
产品负责人询问团队其余成员的意见。



由此可见，仅仅因为A先生最先说出来，就严重影响了团队其余成员的判断。B女士和C女士或许认为工作量应当是中或大，我们就没有机会听到她们的解释，而这恰恰有可能是很有价值的信息！

19.2 用规划扑克进行估算

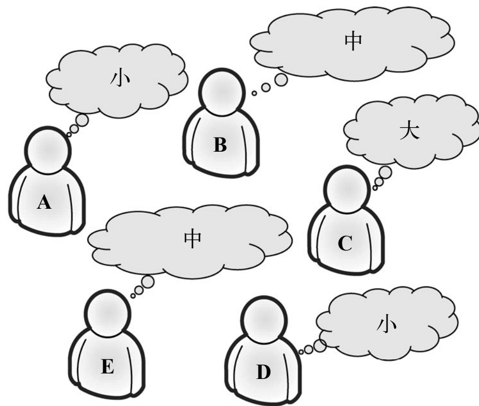
现在想象一下，每个团队成员都拿着下面这些牌。（我们的规划扑克实际上用的是数字：1代表小，2代表中，3代表大。）



我们来重新估算一下。产品负责人说……

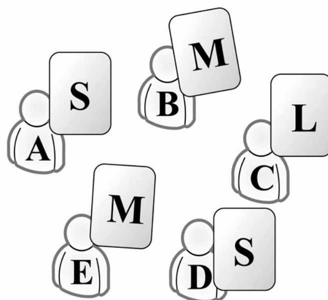


团队成员再一次开始思考这项功能有多大。



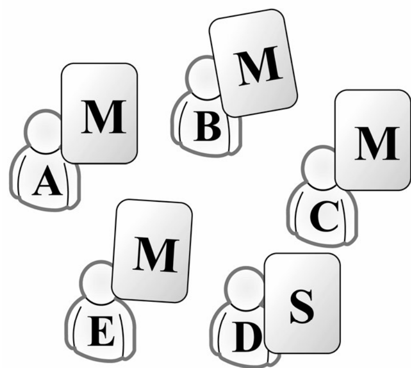
这次没有人不假思索就说话。他们都需要出一张牌，给出自己的估算结果，不过要牌面朝下。人人都需要出牌，于是 D 先生和 E 先生醒了。D 先生承认他睡着了，问大家在估算什么。用这种方式进行估算就没法再偷懒了！

等他们出完牌后，所有牌要同时翻开，显示每个人的估算结果。



哎哟！分歧好大。团队成员，尤其是 A 先生和 C 女士，需要讨论一下这项功能，为什么他们的估算结果会有很大分歧。经过讨论，A 先生意识到他忘记这项功能还需要完成一些很重要的任务。C 女士则意识到按照 A 先生的设计，这项功能可能没那么大。

经过讨论之后（总共三分钟），他们对这项功能再进行一轮估算。



意见一致了！好吧，不完全一致。但他们都同意中这个估计不错。下一项功能！

19.3 特殊牌



问号牌的意思是："我完全没概念。这项功能可能很大，也可能很小。"这种情况应该很少出现。如果这张卡用得很频繁，团队就需要再好好讨论一下这项功能，让团队成员多了解一些信息。



咖啡杯牌的意思是："太累了，脑子卡住啦。我们休息一下吧！"



小乔爱问：

只要估算出现较大分歧，就平均估成中不可以吗？

不可以，如果团队讨论了分歧原因，然后在此基础上重新开始一轮估算的时候就不可以。某位成员可能会说服其他成员某项功能应该为小，因为该功能已经在另外一个系统上实现，可以直接重用。或者反之，有位成员说服了团队其他成员某项功能应该为大，因为存在一些大家都没有考虑到的风险。

多数团队都喜欢用规划扑克来进行估算工作，因为这样不但少了很多痛苦，还能让估算过程变得简单有趣。规划扑克最大的价值在于玩牌时引发的对话与沟通，而估算结果本身则只是顺带来的好处。

第20章

因 果 图

因果图是以图解方式进行因果分析的一种简易、实用的方法。多年来，我一直在用因果图帮助客户了解并解决技术问题以及组织方面的各种问题。

下面我将详细介绍因果图的工作原理，以方便读者根据自己的实际情况予以运用。

20.1 解决问题，而不是解决症状

有效解决问题的关键是，首先确保自己理解了要解决的问题——为什么需要解决，问题得到解决时如何知道这一点，以及问题的根本原因在哪里。

症状一般出现在某处，而根本原因则在另一处。如果“头疼医头，脚疼医脚”，解决了症状但未予以深究，那问题就很有可能会以另一种形式再次出现。我们来看几个例子。

- 问题：卧室里冒烟。
  - 糟糕的解决方案：打开窗户，回去继续睡觉。
  - 好的解决方案：找到哪里在冒烟并解决问题。哎哟，地下室着火了！先赶紧灭火，然后找到起火原因，安装火警警报器，下次再起火就可以预先报警。
- 问题：前额发烫，好累。

  - 糟糕的解决方案：在前额上敷冰块降温。喝点咖啡提神，继续工作。
  - 好的解决方案：量一量体温。哟，发烧了！回家休息。
- 问题：服务器内存泄漏。
  - 糟糕的解决方案：买新内存。
  - 好的解决方案：找到内存泄漏的根源并修复问题。进行测试，检测是否存在新的内存泄露问题。
- 问题：船里进水。
  - 糟糕的解决方案：把水抽出去，继续划船。
  - 好的解决方案：找到水从哪里进来的。啊，有个洞！先补好洞，然后再把水抽出去。

……诸如此类。

企业组织中的多数问题都是系统性的。“体系”（你所在的企业组织）有点小问题，需要修复。在没有找到问题的根源之前，所有希翼解决问题的尝试都是徒劳的，甚至会适得其反。

20.2 精益问题解决方法：A3 思维

精益思维的核心宗旨之一是改善——持续改善流程。丰田将成功主要归功于其高度严谨的问题解决方法。该方法有时称为 A3 思维（以一张 A3 大小的纸为模板，总结解决问题的方案）。

你可以从 http://www.crisp.se/lean/a3-template 下载 A3 示例和模板。这里不太方便显示，因为 A3 纸比我们的书要大很多。不过，图 2 为一个 A3 示例，可以让你对 A3 模板有个大体印象。

A3 Problem Solving Template, Example, and Assessment Questions - version 1.1 - By Tom Poppendieck and Henrik Kniberg.

Owner: Lisa
Mentor: Heinrich
Date: 18 May 2009

Background

Games out of date

⇒ Missed market windows-Revenue is declining
⇒ Demotivated teams-Key developers about to quit
⇒ Overhead costs -Time to develop games steadily increasing due to declining technical quality
⇒ Pressure to Work FASTER!

Current Condition

Game backlog (8) · Design-ready games (15) · Product-ready games (12)

Sam → Concept pres. → Lisa assigns resources → Graphics design → Sound design → Dev → Integr. & deploy

Waste: 2d, 1m, 6m, 1w, 6m, 6m
Value: 2h, 4h, 1d, 1m, 3w, 2m, 1m (Total 3m), 3w

- Process cycle efficiency = 3 months add value / 25 months cycle time = **12%**

Goal / Target Condition

- 8x faster cycle time
- 5x fewer escaped defects
- 20% improvement in revenue

Root Cause Analysis

Key Engineers about to Quit
Sales Revenue Declining
Competitive Market
Engineers NOT proud of their work
Games Stale – Out of Date
Pressure from Business to Work Faster
Company has not grown people to vet game ideas
Founder/CEO no longer has time to vet new game ideas
Game Quality Declining
Taking ever longer to complete a game
Benchmarking > copying Competitors
Defects Tolerated
Technical Debt Increasing
Endless Delays, Thrashing
No Time to Refactor
Too many new game ideas
Queues
No effective selection Filter
Teams focus on their own parts
Work in Progress Exceeds Capacity
Weak understanding of Market wants
Features cannot be built by single team X-team dependencies
Teams divided by Architecture
No unified view of priorities
Engineers don't get to play games

Countermeasures

1. **Cross Functional Teams - Graphics design through deployment**
   ✓ Predict 2x Faster Delivery
      ⇒ End dependencies - now spend 75% of time waiting/negotiating
2. **Abandon all but most promising 3 games in each queue. Do ONE game per cross functional team at a time.**
   ✓ 4x faster delivery from reduced task switching
   ✓ Eliminating queues will cut 1.3 years from schedule
3. **Engage developers in playing games and selecting ideas**
   ✓ 30% more profit to par with best competitor
      ⇒ Improved filtering on which games to develop
      ⇒ More fun games, more popular

Confirmation (Results)

1. **Cross Functional Teams**
   ⇒ Half as much time waiting
2. **One game at a time**
   ⇒ Queues eliminated, time to complete game is 4 months (6x)
   ⇒ Technical Debt decreasing - Escaped defects down by 2x so far
3. **Engage developers in playing games and selecting ideas**
   ⇒ One team taking time to play is producing more innovative games
   ⇒ Impact on profit is TBD.

Follow-up

1. Consider more cross training of team members to reduce waiting for expertise
2. Reduce difficulty of integration and deployment steps
3. Improve processes for generating and selecting game ideas
   a. Recruit talent ifidentifiable/available
   b. Improve skills/process of best people already in company
   c. Broaden both participation in selection and game playing experience of everyone in the company.
4. Continue improvement of reused game components/engines to improve development throughput and reduce defects.

图 2　A3 方法示例

采用 A3 方法时，在提出解决方案之前，我们需要花费大量时间（图中左边部分）对问题的根本原因进行分析并以图示形式显示。因果图只是用来进行因果分析的一种方法。还有其他方法也可以使用，如价值流图和鱼骨图。这个 A3 示例里用到了价值流图（左上）和因果图（左下）。

因果图的优点是相当直观而且无需解释（尤其是与鱼骨图相比而言）。另一个优点是你可以显示增强回路（恶性循环），这从系统思维角度而言非常有用。

下面我们来看看如何有效地创建并使用这些图。

20.3 如何使用因果图

使用因果图的基本流程如下。

(1) 选定问题，困扰你的任何问题，并写下来。

(2) “向上”追踪，弄清业务后果，即你的问题所造成的“可见损坏”。

(3) “向下”追踪，找到根本原因。

(4) 确定并标出恶性循环（循环路径）。

(5) 将这些步骤重复几次，调整并理清因果图。

(6) 决定要解决哪些根本原因，以及如何解决（即要实施哪些对策）。

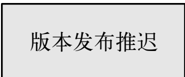
之后进行跟进。如果实施的对策起作用，那么值得祝贺；如果不起作用，也别气馁。分析一下为什么不起作用，根据分析得来的经验教训更新因果图，然后尝试其他对策。

一项对策就是一次实验，而不是一个解决方案。你假设这项对策会解决（或减轻）问题，但不能保证。实际上，你是在试探整个体系看它如何反应。这就是为什么跟进工作很重要。

失败只意味着企业的体系在试图向你传递信息——所以你要仔细聆听。唯一的真正失败是未能从失败中吸取经验教训！

20.4 示例 1：发布周期长

假设我们的问题是总是赶不上最终期限。具体来说，就是我们的版本发布总是比预定计划晚几天。

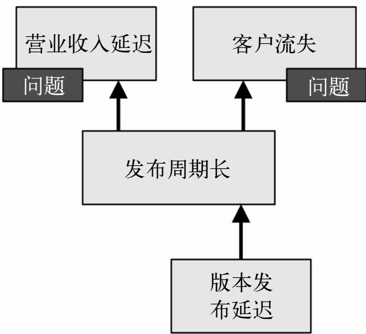


只有与我们的目标有冲突的问题才是问题。所以，从定义目标开始，思考这个问题从目标角度而言有什么后果。具体做法是通过回答几个“那又怎么样？”的问题来确定可见损坏。

假设我们的目标是让客户高兴，让我们的营业收入最大化。我们的对话可能是下面这样。

丽莎：如果版本发布延迟，都有谁会介意？有什么后果？
吉姆：延期发布会把我们的发布周期拖长。
丽莎：那又怎么样？
吉姆：那样就会拖延我们的营业收入，然后会影响到现金流。还会让客户流失，因为他们的耐心有限，不愿无故多等！

我们一边交谈，一边往因果图上添加方框和因果箭头。通常在试图确定后果的时候，我都会从原始问题开始“向上”追踪，不过并不需要严格遵循这条规则。



从图中我们可以看到，版本发布延迟并不是真正的问题。真正的问题是营业收入延迟和客户流失。这个时候我们需要考虑以下三个方面。

- 是否还有其他问题造成客户流失或营业收入延迟？如果是这样，版本发布延迟是主因吗？还是我们应当把注意力放在其他地方？
- 我们能把问题量化吗？我们流失了多少营业收入？流失了多少客户？我们可以利用这些数据来评估该花多少精力来解决这个问题。
- 问题若得到解决，我们如何能够知道？如果我们请的咨询专家来了以后搞出很大动静，然后他骄傲地宣布“我已经解决问题了”，我们如何才能揭穿骗局？

等我们分析清楚了问题的后果，就该向下挖掘，找到根本原因。

首先，多问几个“为什么”。对，这就是精益思维的“五个为什么”方法。如果你学习过精益思维，就可能已经听说过这一方法。

丽莎：版本发布为什么会延迟？

吉姆：因为范围在不断增加。

丽莎：为什么？

吉姆：因为客户提出要增加新功能，他们坚持要加到现在这个版本里，而且拒绝拿掉优先级低的功能。

丽莎：为什么？为什么不把那些功能递延到下一个版本？

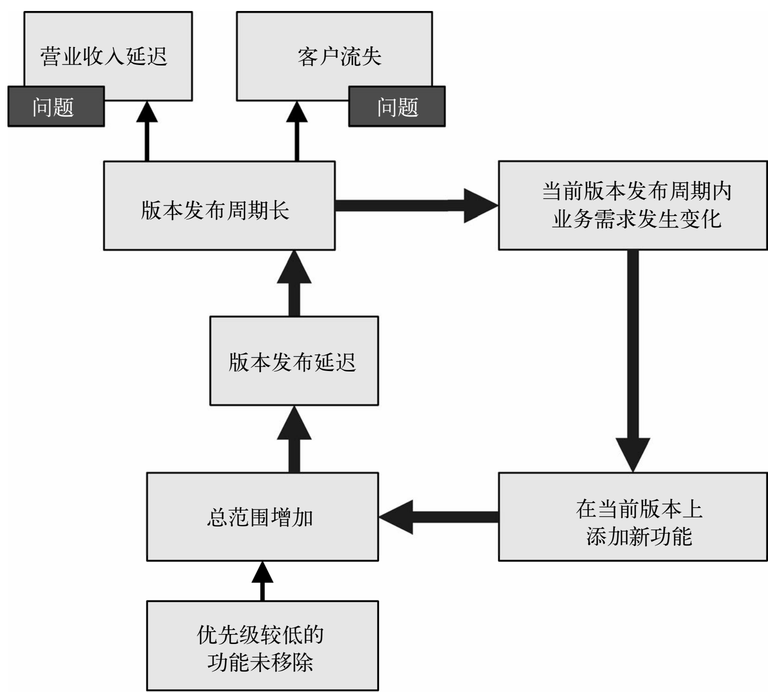
吉姆：因为版本发布周期太长，一个版本还没发布，就出现了新的需求！

这里只有三个为什么。不过你应该对“五个为什么”方法有点概念了。

根据吉姆和丽莎的对话内容，我们得到下面这张图。

恶性循环（或增强回路）用粗箭头突出显示。重复发生的问题几乎无一例外都有这样的循环问题，但可能需要花点时间才能找出来。找到这种循环就大大提高了切实而永久解决问题的可能性！

我们的目标是确定问题的根本原因，这样就能花最少的精力达到最大的效果。第一关很容易错过重要的原因，所以我们要返回去再多问几个为什么。



丽莎：版本发布周期为什么太长？版本发布延迟是唯一的原因吗？

吉姆：嗯，实际上，就算发布不延迟，我们原先计划的版本发布周期也相当长。

丽莎：原先计划的版本发布周期有多长？

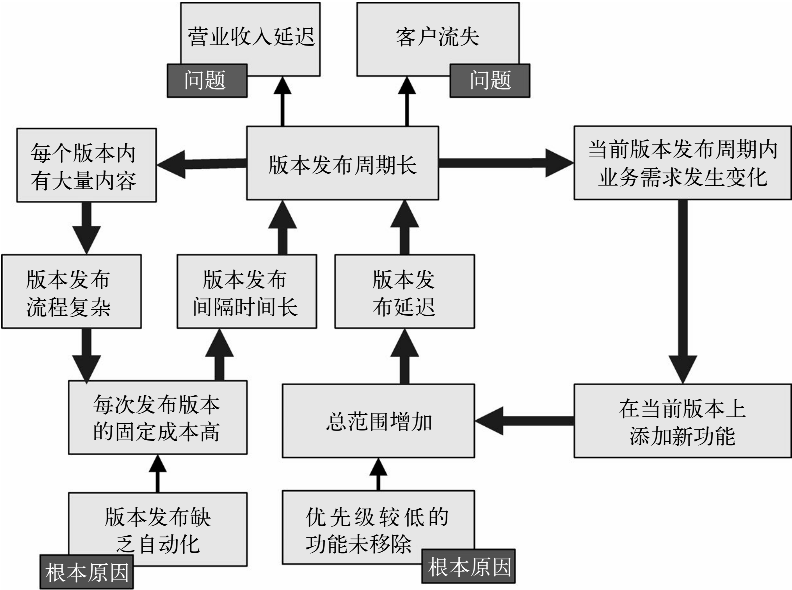
吉姆：一个季度一次。

丽莎：那为什么会这么长？

吉姆：因为版本发布成本很高，而且很复杂。

丽莎：为什么？

吉姆：因为每个版本都有太多工作要做，而且都是手工的。



看看左边——又一个恶性循环（粗箭头）！版本发布间隔时间长意味着每个版本都包含大量内容，这就意味着难度大、成本高，让我们不情愿提高发布频率。

如图所示，我标出了两条根本原因。现在该制定对策了。

根本原因	对　策
版本发布缺乏自动化	实施版本发布自动化
优先级较低的功能未移除	与客户谈判，定下一条规则，只有在移除相应大小的优先级较低的功能，才允许他们往当期版本中增加新功能

在确定哪个小问题是根本原因方面并没有严格的规则，不过可以参考下面的这些指标。

- ❑ 某个小问题只有输出方向的箭头，而没有输入方向的箭头。
- ❑ 从这里开始继续往下挖（多问几个更进一步的“为什么”）已经没有什么意义。
- ❑ 这个小问题是我们能够解决的，而且对我们希望解决的问题可能会有积极作用。

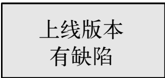
五个“为什么”方法之所以得名，是因为通常问了五个“为什么”就能找到根本原因。我们容易过早结束提问，所以要继续向下挖！

注意，我们最初确定的问题——版本发布延迟——并不见得是真正的问题或根本原因，那只是症状。我们以此为契机向上挖可以确定真正的问题，然后向下挖可以确定根本原因。这种做法能够让我们在获得足够信息的基础上提出切实可行的对策。

没有这种类型的分析，我们就可能会草率得出结论，并实施无效的甚至是适得其反的变更——例如，增加更多人手，虽然人手多少跟我们要解决的问题没有任何关系；或者改变激励模式（若按时发布，则予以奖励；若延迟发布，则予以惩罚），即便当前的激励模式与问题没有任何关系。我敢打赌你一定多次见过这样的情形。

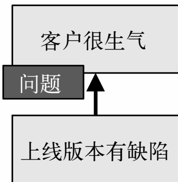
20.5 示例 2：上线版本有缺陷

假设我们刚刚发布上线的版本中有代码缺陷问题。



丽莎：那又怎么样？

吉姆：缺陷把客户惹恼了！

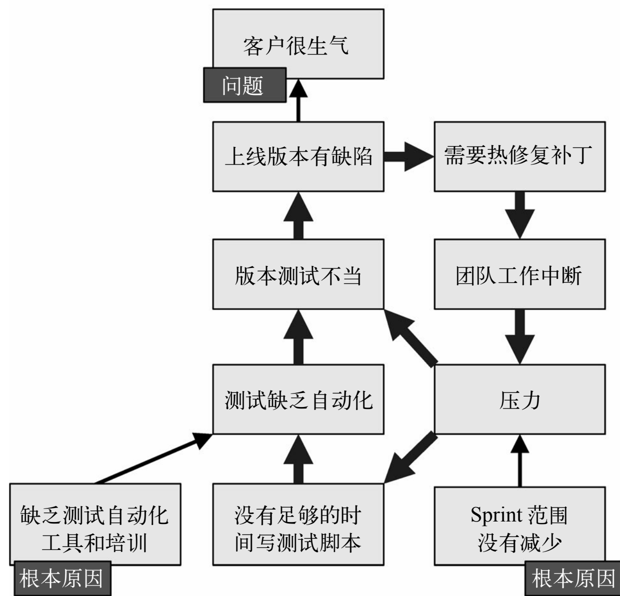


丽莎：发布上线的版本为什么会有缺陷？

吉姆：因为发布之前测试不当。

丽莎：为什么测试不当？

……诸如此类。然后我们就得出下图。



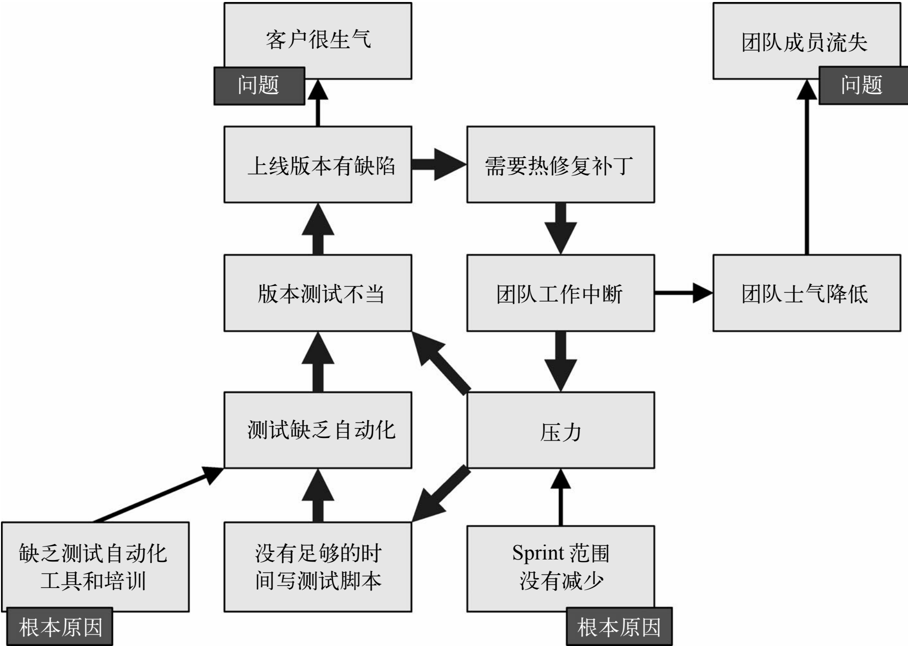
看看吧，两条增强回路！看一看箭头。

回路1（内圈）：产品中的缺陷导致需要热修复补丁，这就中断了团队手中的工作。而且因为不允许团队缩减项目范围，大家压力倍增，所以就没有时间完整测试新版本。而这当然就会导致上线版本中出现更多缺陷。

回路2（外圈）：因为大家都处于压力之下，也就没有时间写自动化测试脚本。这就导致测试自动化整体缺乏，要完成新版本的回归测试就越来越困难，而这又导致上线版本中存在缺陷，于是需要更多的热修复补丁，最终带来更多压力。

别急，还有更多问题！

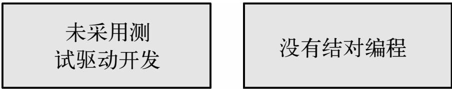
开发团队讨厌手中的工作被打断。这样就打断了流程，并从长期而言摧毁了团队的士气。这或许可以解释为什么人员流动率一直居高不下！所以，在解决原始问题（上线版本存在缺陷）的同时，我们还得到额外收获，即团队的人员流动率降低了！



这就是解决根本原因的优势。根本原因通常会导致多个问题（所以把它称为根本原因）。

20.6　示例 3：缺乏结对编程

曾有客户要我帮忙搞清楚他们为什么没有采用极限编程的做法，如结对编程和测试驱动开发。客户说："我们都清楚应该这样做的，但实际上没有。"



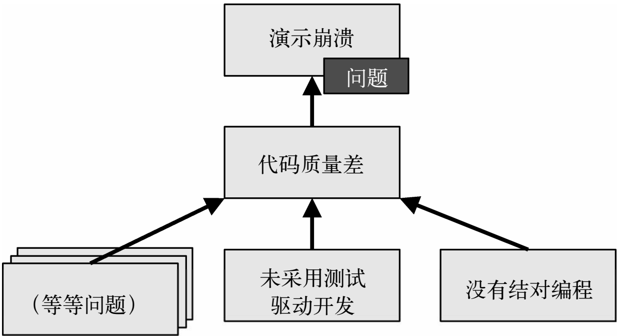
那么，未采用测试驱动开发（TDD）和结对编程的做法真的是问题吗？照例，我们称之为问题的通常都只是症状而已。

丽莎：不做结对编程和 TDD 的后果是什么？

吉姆：如果采用了这两种做法，我们的代码质量应该会更高。

丽莎：代码质量差有什么后果？你碰到过因为代码质量差造成的什么实际问题吗？

吉姆：有，我们有几次演示崩溃了。我们是家研究型公司，要通过演示来赢取订单，所以这个问题比较严重。



我们以其中一个小问题为例，看看是否能够向下挖掘到根源。

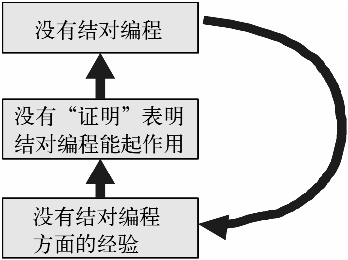
丽莎：那么，你们为什么没有采用结对编程的做法？

吉姆：因为很多人都担心结对编程不起作用，怕因此浪费时间。我们没法证明结对编程能够起作用。

丽莎：你们需要什么样的证明？

吉姆：嗯，我们看到有研究表明这种做法有用。但这里没人有这方面的经验，所以我们不确定这种做法的实际效果如何。

这里是第一个回路。



他们因为不知道结对编程的做法能否起作用，所以就没有采用。他们不确定是因为没有这方面的经验。

丽莎：为什么你们没有至少尝试一下结对编程呢？

吉姆：我们没时间去尝试。

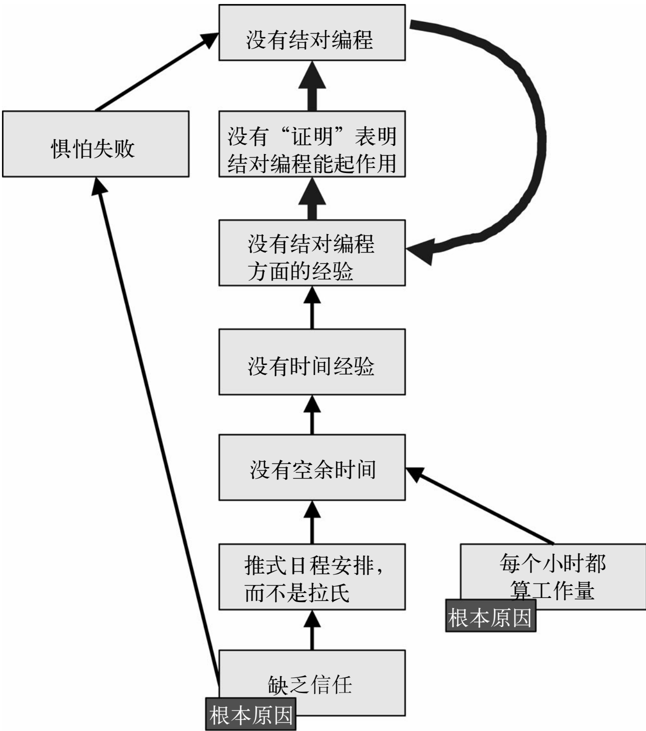
丽莎：为什么没时间？

吉姆：因为我们没有空余时间，所有时间都安排满了。客户还在不断增加我们的工作量。

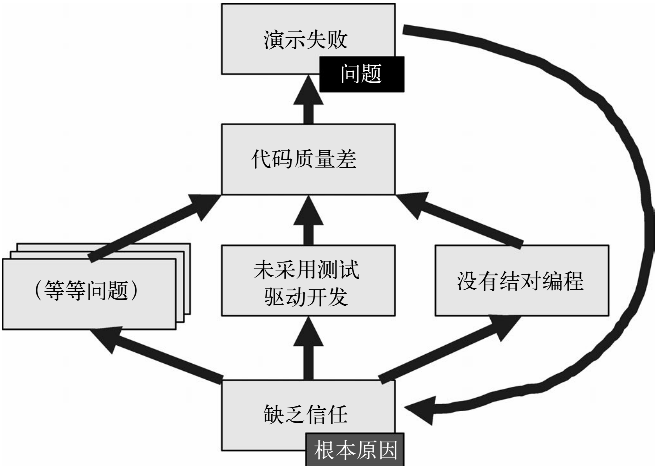
丽莎：他们为什么不让你们自己管理工作时间？这样你们只要有空，不就可以自行决定接手更多的工作了吗？

吉姆：他们不相信我们能够有效地利用自己的工作时间。

缺乏信任也导致惧怕失败，而这就显然让他们在没有“证据”的情况下不愿意去尝试新方法，如结对编程。



看来有两大根本原因：缺乏信任与所有工作时间都必须有产出的管理原则。我们把根本原因放回整体情形中再看看。



我们发现，缺乏信任是导致团队没有遵循 XP 实践（如测试驱动开发和结对编程）的根本原因，而这则导致质量差，继而导致演示失败。猜猜怎么着？演示失败又更进一步降低了信任度。这里存在一个恶性循环！

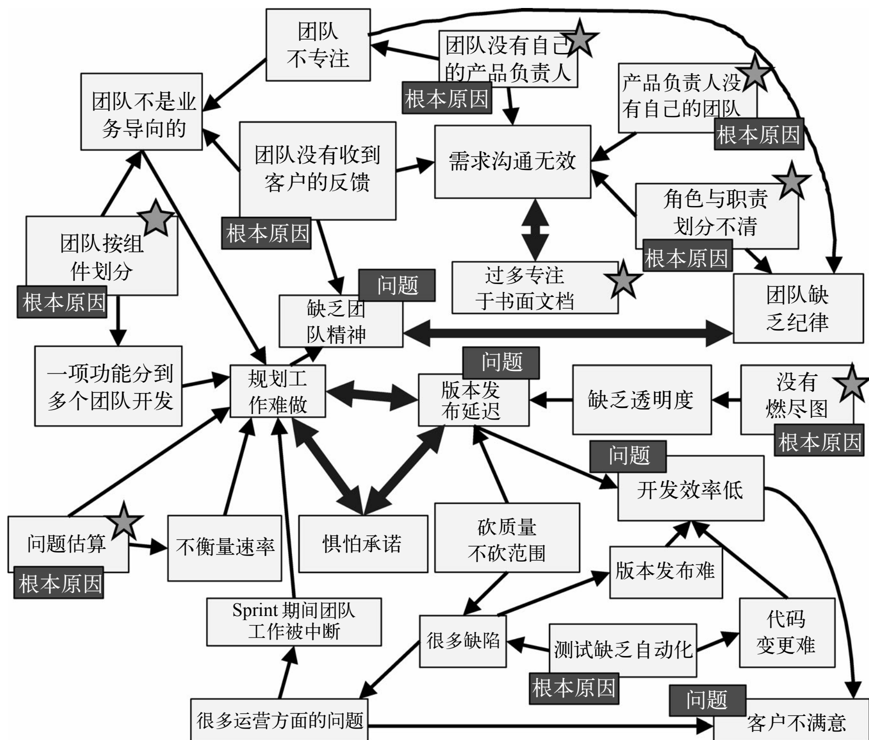
我们在一个为期两天的研讨会上与 25 个人一起做了这些练习。开始的时候，我们讨论的主要是技术——如何开始实施测试驱动开发和结对编程等，不过未见成效。于是，我们将大家分成几个小组，每个小组选一个问题，画因果图，并创建解决问题的 A3 方案。有意思的是，其中几个小组分析的问题完全不同，但发现的根本原因却一模一样：缺乏信任！上面最后一幅图只是其中一个例子。

当天结束时，我们已经在讨论能做什么来提高客户与开发人员之间的信任程度，这种话题转变让人惊讶。

作为第一步，我们一致认为，下一次召开这种类型的研讨会时应当邀请“他们”（客户）一起参加，这样就能少用诸如“我们”和“他们”等词汇。

20.7 示例 4：很多问题

下面是一个更大规模的示例。这个组织采用 Scrum 开发模式，但碰到一些问题。访谈和研讨会之后得出的因果图表明，他们并未遵循正确的 Scrum 实践，正是这一点导致了诸多问题。



现在大家都很清楚，很多根本原因可以通过“正确”实施 Scrum 实践得到解决（例如，将整个团队重组为多个跨职能团队，并确保每个团队都有一位专职的产品负责人）。这种思路引发了组织变化，并最终修正了很多根本原因（星星）。下一步是改进测试自动化。

当然，Scrum 解决不了所有问题。实际上，有时 Scrum 本身就有问题，而其他一些技术（如看板）才是解决方案。有关 Scrum 与看板如何完美结合使用，请参见 *Kanban and Scrum— making the most of both*[KS09]一书。

20.8 实际问题：如何创建并维护因果图

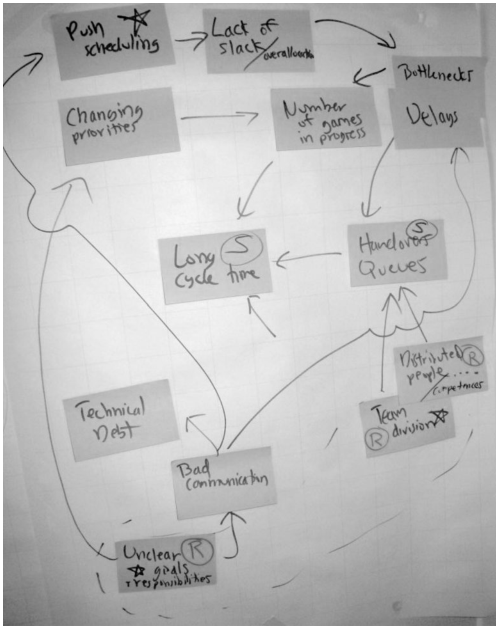
那么，我们应该如何创建因果图？嗯，这部分取决于有多少人参与。

单独工作

如果是独自一人创建因果图，我发现直接运用绘图工具（如 Visio 或 PowerPoint）工作最为便捷，因为这样画图时，能够快速移动图表元素、调整图框并快速进行备份。

与一小组人一起工作（2~8 人）

大家聚集在白板或挂图前。用便利贴来写下问题，然后画箭头把便利贴连起来。建议采用白板，这样就可以在移动便利贴时擦除老箭头，然后画新箭头。确保人人都在参与，而不是只有一个人负责画图。确保有人负责拍照，把内容都照清楚，以便会后发给所有人。



与一大组人一起工作（9~30 人）

把大家分成几个小组，然后每个小组专注于一个具体问题。让多个小组分别独立研究同一个问题的做法是可以的——他们会得出相同或不同的结论，两种情况都会很有意义。每个小组都单独运用挂图/白板和便利贴。然后，定期将所有人都聚拢到一起，让大家分享各自的研究心得。

因果图的长期维护

可以直接用 Visio 或 PowerPoint 等绘图工具来维护因果图。如果组织了研讨会形式的会议，先决定好会议目的是展示因果图还是更新因果图。如果是展示因果图，就用投影仪直接将 Visio（或你使用的其他绘图工具）中的因果图投影到大屏幕上。如果是更新因果图，就把图复制到白板/挂图上，包括便利贴和所有箭头，这样可以让大家高效协作。然后，会议结束后，将所有变动更新到电子版的因果图上。

这种方式的同步需要花点时间，不过通常都很值得。开团队讨论会时，没有什么能比实体白板和便利贴更见成效的了。

20.9 陷阱

下面介绍一下创建因果图的一些常见陷阱以及如何避免这些陷阱。

太多箭头和图框

有时原本很有用处的因果图会像老鼠迷宫一样杂乱不堪，这时需要简化因果图。可以参考下面这些简化技巧。

- 移除多余的图框（不能给因果图增加价值的图框）。
- 专注于“深度优先”，而不是“广度优先”。别把问题的所有原因都写下来；只写最重要的一到两个原因，然后不断向下挖掘。

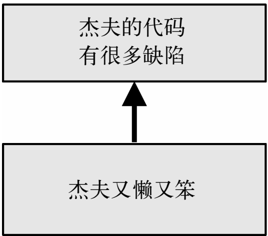
- 接受不完美：这种图永远都不可能完美。乔治·博克思（George Box）说得非常好："所有模型都是错的，不过其中一些还是有用的。"
- 你的问题涉及的区域有可能非常广。尝试将自己限定在范围很明确的问题上。
- 将因果图划分成若干部分，就像我在示例 3 中的做法那样。

过于简化

这种因果图的意图就是保持简单。它替代不了面对面的沟通。如果你需要更加高级或更加正式的工具，可以阅读有关系统思维方面的书，如《第五项修炼》（*The Fifth Discipline* [Sen94]）等。这些书讲述了如何区分增强回路和平衡回路，以及如何添加时间维度（显示存在延迟的情况下 X 如何导致 Y）。不过要注意：如果需要有博士水平才能理解，即使再"完美"的图都是没有用处的。

掺杂个人情绪

要避免"谴责游戏"问题，如下图所示。



如果假定所有问题都是系统问题，问题解决的效果就最好。当然，的确有人比较笨。不过即便这是导致大问题的原因，那也仍然是系统问题，比如，我们的系统假定笨人不笨，我们的系统让超级笨蛋都能进来，我们的系统没能帮助笨人变得没那么笨，诸如此类。

20.10 为何采用因果图

简而言之，因果图具有如下强大优势。

- 建立共识：以团队为基础解决问题极为有效，但需要团队所有人员对问题拥有共识。因果图是非常实用的协作技术。
- 发现问题对业务的影响：了解了这些能让大家首先专注于最重要的问题，并作出知情决策。
- 找到根本原因：这有助于强化变更效果。
- 消除恶性循环（负面增强回路）：打破恶性循环或将其转变为积极的增强回路（好的方面引出更好的结果，而不是不好的方面导致更差的结果）。

因果图虽然有用，关键却是解决问题的方法本身：提问的角度、由此展开的讨论以及调查发现。你甚至不需要真的把图画出来，只要在谈话的过程中在脑中想象一下就足够。

这种解决问题的结构化方法在几乎任何情景下都很有用——例如，开导你的朋友或改善自己的生活。亦或，甚至是改进这个世界！

第21章

结　语

好了，这段很长的旅程要结束了。现在我们就来总结一下！

本书第一部分介绍了我们的项目——我们如何工作，以及在项目中我们学到了什么经验教训。第二部分介绍了我们在项目中用到的一些具体技术。然后呢？

如果你希望了解更多的信息，可以关注我的博客：http://blog.crisp.se/author/henrikkniberg。你也可以前往这个网址，对本书提出反馈或参与讨论：http://pragprog.com/book/hklean/lean-from-the-trenches。

我真想给你列出一个很长的参考书目，不过我不会这样做。目前你已经读到了够多的信息，是时候放下书本回到实战中！

只有经过实践证明的知识才能站得住脚。所以，思考一下你从这本书中学到了什么，以及你会如何将这些知识运用到自己的项目中。然后，尽管放手实验！

附　录

术语表：如何避免高深术语

对于普通人，尤其是非英语国家的人而言，精益与敏捷的很多词汇听上去都很奇怪。（PUST 项目所有成员都讲瑞典语。）诸如产品需求清单（product backlog）、回顾总结（retrospective）、用户故事（user story）、速率（velocity）、Scrum 大师（Scrum master）和故事点（story points）等词汇，足以使外行退避三舍。

所以，我在项目中尽可能使这些词听上去不那么高深，因为没有必要让大家敬而远之。事实证明，谨慎选择术语非常很有帮助，那就让我跟大家分享一下我们的术语表。

显然，这部分与瑞典语读者最为相关。

瑞典术语	英语术语	字面意思	实际含义（对应的说法）
Processförbättringsmöte	Process improvement meeting	流程改进会议	Sprint回顾总结 （Sprint retrospective）
Leverabel	Deliverable	可交付物	功能（feature），产品需求清单项（product backlog item）
Kundleverabel	Customer deliverable	客户可交付物	用户故事（user story），与技术故事和其他非客户相关工作相对
Funktionsområde	Feature area	功能区域	综述（epic），主题（theme）
Teamledare	Team lead	团队主管	Scrum大师（Scrum master）
Projekttavla	Project board	项目看板	项目级别的看板
Teamtavla	Team board	团队看板	团队级别的看板/Scrum混合板

术语可交付物（leverabel）尤为有用。在此之前我们用 krav（=需求）来表示所有相关信息。现在，可交付物（leverabel）与功能清单（funktion-område）之间则有了明显区别。

Lean from the Trenches
Managing Large-Scale Projects with Kanban

精益开发实战
用看板管理大型项目

“我从头到尾读完了这本书。一句话，太棒了！扎实、真切、有趣、易读、流畅，极好地平衡了理论与实践。”

—— 肯特·贝克（Kent Beck），敏捷先驱，著名软件工程师、作家

“卓越之作。向作者致敬！你不辞辛劳地记录下了大型项目获得成功需要做出的各种日常决策。希望这本书会成为判断众多项目成功与否的基准。”

—— 沃德·坎宁安（Ward Cunningham），维基概念发明者，设计模式和敏捷软件方法先驱

“这本书让我们看到了一个将敏捷流程的精髓应用于真实场景的卓越案例。如果你曾困惑于‘我做得对不对’之类的问题，这本书就会告诉你答案。所有技术团队主管，只要想了解敏捷流程如何实际运作，都应该现在就买这本书！”

—— 科林·耶茨（Colin Yates），英国QFI咨询公司首席工程师

精益、看板、Scrum这些敏捷术语很时髦，但这些东西究竟怎么样，是否实用，如何具体应用，很多人并不了解。

本书以瑞典国家警署开发的大型项目为例，讲述在大型项目中如何具体应用看板方法和精益原则，详细介绍了项目中面临的诸多挑战及其应对策略，以及得到的各种经验教训。书中内容共分为两大部分，第一部分是全书核心，介绍如何实际工作；第二部分是技术讲解，概要介绍了敏捷和精益原则，阐述第一部分提到的因果图等实践做法。

精益是方向，而不是目的，其核心是持续改进。谨以此书献给所有有兴趣了解软件开发、精益产品开发和日常协作技术的读者，更希望它能帮助软件开发组织中的项目团队主管、经理、教练和其他负责人成功实施项目。

The Pragmatic Programmers

图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐邮箱：contact@turingbook.com
热线：（010）51095186转604

分类建议 计算机/敏捷

人民邮电出版社：www.ptpress.com.cn



ISBN 978-7-115-29177-6
定价：39.00元